Proceedings of the Young Researchers’ Conference “Frontiers of Formal Methods”

Thomas Ströder and Wolfgang Thomas (Editors)
The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/
Contents

Preface .................................................... 11
Program Committee ................................. 13

Invited Speakers ........................................ 15
Moshe Vardi (Houston): The Rise and Fall of
Linear Temporal Logic ............................... 17
Bernd Finkbeiner (Saarbrücken): Distributed
Synthesis ................................................. 19
Jean-Francois Raskin (Brussels): Variations
on the Stochastic Shortest Path Problem ....... 21
Azadeh Farzan (Toronto): Succinct Proofs of
Concurrent Programs ............................... 23
Eric Bodden (Darmstadt): SPLift: Statically
Analyzing Software Product Lines in Minutes
instead of Years ...................................... 25
Joel Ouaknine (Oxford): Termination of Lin-
ear Loops: Algorithmic Advances and Chal-
lenges .................................................... 27

Contributions ........................................... 29
Mohamed Abdelaal: Fuzzy Compression Ren-
ement via Curvature Tracking ..................... 31
Parvaneh Babari: A Nivat Theorem for Weighted
Picture Automata and Weighted MSO Logics 37
Stephan Barth: Deciding Monadic Second Order Logic over omega-words by Specialized Finite Automata ........................................ 41
Félix Baschenis: From sweeping transducers to one way transducers ........................................ 47
Benedikt Brütsch: Synthesizing Structured Reactive Programs via Deterministic Tree Automata ........................................ 53
Claudia Carapelle: Satisfiability of ECTL* with constraints ........................................ 57
Pierre Carlier: Composition of Stochastic Timed Automata ........................................ 61
Florian Corzilius: SMT-RAT: An SMT-Compliant Nonlinear Real and Integer Arithmetic Toolbox 67
Christian Dehnert: Fast Debugging of PRISM Models ........................................ 73
Manuel Eberl: A Verified Compiler for Probability Density Functions ........................................ 77
Andreas Ecke: Relaxing Description Logics Queries using Similarity Measures ........................................ 83
Björn Engelmann: Formally Verifying Dynamically-typed Programs like Statically-typed Ones A different perspective ........................................ 87
Shiguang Feng: Path-Checking for MTL and TPTL over Data Words ........................................ 93
Oliver Fernández Gil: Threshold Concepts in a Lightweight Description Logic ........................................ 97
Nils Erik Flick: Derivation Languages of Graph Grammars and Correctness ........................................ 99
Yang Gao: Decision Procedure for Stochastic Satisfiability Modulo Theories with Continuous Domain ........................................ 103
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manuel Gieseking: Trace Refinement of pi-Calculus Processes</td>
<td>109</td>
</tr>
<tr>
<td>Friedrich Gretz: Conditioning in Probabilistic Programming</td>
<td>115</td>
</tr>
<tr>
<td>Dennis Guck: Markov Reward Automata in Railway Engineering</td>
<td>121</td>
</tr>
<tr>
<td>Doreen Heusel: Weighted Unranked Tree Automata over Tree Valuation Monoids</td>
<td>127</td>
</tr>
<tr>
<td>Philipp Hoffmann: Negotiations as a concurrency primitive: Summaries and Games</td>
<td>131</td>
</tr>
<tr>
<td>Johannes Hölzl: Probability Theory and Markov Processes in Isabelle/HOL</td>
<td>135</td>
</tr>
<tr>
<td>Fabian Immler: Continuous Systems Reachability using Adaptive Runge-Kutta Methods - Formally Verified</td>
<td>141</td>
</tr>
<tr>
<td>Christina Jansen: Generating Abstract Graph-Based Procedure Summaries for Pointer Programs</td>
<td>145</td>
</tr>
<tr>
<td>Benjamin Lucien Kaminski: Analyzing Expected Outcomes and (Positive) Almost-Sure Termination of Probabilistic Programs is Hard</td>
<td>157</td>
</tr>
<tr>
<td>Edon Kelmendi: Two-player shift-invariant and submixing stochastic games are half-positional</td>
<td>163</td>
</tr>
<tr>
<td>Saifullah Khan: Traffic Data Dissemination in Realistic Urban VANETs Environment</td>
<td>167</td>
</tr>
<tr>
<td>Veronika Loitzenbauer: A Hierarchical Sparification Technique for Faster Algorithms in Graphs and Game Graphs</td>
<td>173</td>
</tr>
<tr>
<td>Bogdan Mihaila: Synthesizing Predicates from Abstract Domain Losses</td>
<td>177</td>
</tr>
</tbody>
</table>
Sarah Winter: Uniformization of Automatic Tree Relations by Top-down Tree Transducers 265
Sascha Wunderlich: Weight Monitoring with Linear Temporal Logic . . . . . . . . . . . . . . . . . 269

Aachener Informatik-Berichte . . . . . . . . . . . 275
Preface

The Young Researchers’ Conference “Frontiers of Formal Methods” (FFM 2015) is a “singularity” – an event that is not part of a longer conference series, but organized following a nice coincidence of interests of several research groups in Germany and Austria.

It all started with a loose promise given five years ago: When the second phase of the DFG research training group (Graduiertenkolleg) “AlgoSyn” started in Aachen, we promised to organize a final conference at the end of altogether nine successful years of work by and with doctoral students. In discussions between the speakers of closely related research training groups, it became then clear that all our doctoral students would gain most by a jointly organized conference. Five partners joined their forces: the DFG research training groups

- AlgoSyn (Algorithmic Synthesis of Reactive and Discrete-Continuous Systems), Aachen,
- PUMA (Program and Model Analysis), Munich,
- QuantLA (Quantitative Logics and Automata), Dresden & Leipzig,
- SCARE (System Correctness under Adverse Conditions), Oldenburg,
- and the Austrian Research Network ARiSE (Rigorous System Engineering).

AlgoSyn offered to do the local organization, and so all its doctoral students took part in the preparation of the event. Sincere thanks are due to all of them for their diligent
work, as well to Helen Bolke-Hermanns and Silke Cormann for their help in the administration. The program committee was formed from advisors and postdoc researchers of the participating institutions: Erika Ábrahám, Joost-Pieter Katoen, Wolfgang Thomas as chair (Aachen), Franz Baader, Manfred Droste, Karin Quaas (Dresden/Leipzig), Michael Luttenberger, Tobias Nipkow, Helmut Seidl (Munich), Martin Fränzle, Ernst-Rüdiger Olderog, Oliver Theel (Oldenburg), and Roderick Bloem, Martina Seidl, Florian Zuleger (ARiSE).

We are most grateful to our six invited speakers for offering intriguing one-hour lectures: Moshe Vardi (Houston), Jean-Francois Raskin (Brussels), Joel Ouaknine (Oxford), Bernd Finkbeiner (Saarbrücken), Azadeh Farzan (Toronto), and Eric Bodden (Darmstadt).

But the most important contribution to the conference was the work of the doctoral students of the participating institutions and from many other places around the world, condensed into short abstracts as they now appear in these informal proceedings.

A final word of thanks goes to the Deutsche Forschungsgemeinschaft DFG and the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung FWF) for their generous support – and to RWTH Aachen University for providing the infrastructure of FFM 2015.

Wolfgang Thomas
Speaker of the Research Training Group “AlgoSyn”
Program Committee

- Erika Ábrahám, RWTH Aachen University (AlgoSyn)
- Joost-Pieter Katoen, RWTH Aachen University (AlgoSyn)
- Wolfgang Thomas (chair), RWTH Aachen University (AlgoSyn)
- Franz Baader, TU Dresden (QuantLA)
- Manfred Droste, University of Leipzig (QuantLA)
- Karin Quaas, University of Leipzig (QuantLA)
- Michael Luttenberger, TU Munich (PUMA)
- Tobias Nipkow, TU Munich (PUMA)
- Helmut Seidl, TU Munich (PUMA)
- Martin Fränzle, University of Oldenburg (SCARE)
- Ernst-Rüdiger Olderog, University of Oldenburg (SCARE)
- Oliver Theel, University of Oldenburg (SCARE)
- Roderick Bloem, TU Graz (ARiSE)
- Martina Seidl, JKU Linz (ARiSE)
- Florian Zuleger, TU Vienna (ARiSE)
Invited Speakers
Moshe Vardi (Houston):  
The Rise and Fall of Linear Temporal Logic

One of the surprising developments in the area of program verification in the late part of the 20th Century is the emergence of Linear Temporal Logic (LTL), a logic that emerged in philosophical studies of free will, as the canonical language for describing temporal behavior of computer systems. LTL, however, is not expressive enough for industrial applications. The first decade of the 21 Century saw the emergence of industrial temporal logics such as ForSpec, PSL, and SVA. These logics, however, are not clean enough to serve as objects of theoretical study. This talk will describe the rise and fall of LTL, and will propose a new canonical temporal logic: Linear Dynamic Logic (LDL).
Bernd Finkbeiner (Saarbrücken): Distributed Synthesis

More than fifty years after its introduction by Alonzo Church, the synthesis problem is still one of the most intriguing challenges in the theory of reactive systems. On the one hand, synthesis algorithms have found applications in many areas of computer science and systems engineering, from the construction and optimization of circuits and device drivers to the synthesis of controllers for robots and manufacturing plants. On the other hand, the logical and algorithmic foundations of the synthesis problem are still far from complete. In this talk, I will focus on the problem of synthesizing distributed systems, a particularly interesting, and also particularly difficult, version of the synthesis problem. I will give an overview of the state of the art in models, logics, and algorithms for the synthesis of distributed systems and present ideas for future directions.
Jean-Francois Raskin (Brussels): Variations on the Stochastic Shortest Path Problem

In this talk, we revisit the stochastic shortest path problem, and show how results allow one to improve over the classical solutions: we present algorithms to synthesize strategies with multiple guarantees on the distribution of the length of paths reaching a given target, rather than simply minimizing its expected value. The concepts and algorithms that we propose here are applications of more general results that have been obtained recently for Markov decision processes and that are described in a series of recent papers.
Azadeh Farzan (Toronto): Succinct Proofs of Concurrent Programs

In this talk, I will briefly look at the general trends in the history of proof methods for concurrent programs, and the dominant quest for compositional proof methods. I will then talk about the recent progress that my colleagues and I have made in this area. The key observation is that compositionality is not the only way of achieving succinctness in proofs, and decidability or tractability in the verification process.
Eric Bodden (Darmstadt):

SPLlift: Statically Analyzing Software Product Lines in Minutes instead of Years

A software product line (SPL) encodes a potentially large variety of software products as variants of some common code base. Up until now, re-using traditional static analyses for SPLs was virtually intractable, as it required programmers to generate and analyze all products individually. In this talk, however, I will show how an important class of existing inter-procedural static analyses can be transparently lifted to SPLs. Without requiring programmers to change a single line of code, our approach SPLlift automatically converts any analysis formulated for traditional programs within the popular IFDS framework for inter-procedural, finite, distributive, subset problems to an SPL-aware analysis formulated in the IDE framework, a well-known extension to IFDS. Using a full implementation based on Heros, Soot, CIDE and JavaBDD, we were able to show that with SPLlift one can reuse IFDS-based analyses without changing a single line of code. Experiments using three static analyses applied to four Java-based product lines showed that the approach produces correct results and outperforms the traditional approach by several orders of magnitude.
Termination of Linear Loops: Algorithmic Advances and Challenges

Joel Ouaknine (Oxford):
Termination of Linear Loops: Algorithmic Advances and Challenges

In the quest for program analysis and verification, program termination – determining whether a given program will always halt or could execute forever – has emerged as a pivotal component. Unfortunately, this task was proven to be undecidable by Alan Turing eight decades ago, before the advent of the first working computers! In recent years, however, great strides have been made in the automated analysis of termination of programs, from simple counter machines to Windows device drivers.

In this talk, I will focus, from a theoretical (i.e., decidability and complexity) point of view, on the special case of simple linear loops, i.e., un-nested WHILE programs with linear assignments and linear exit conditions (and no conditionals, side effects, nothing). Somewhat surprisingly, the study of termination of simple linear loops involves advanced techniques from a variety of mathematical fields, including analytic and algebraic number theory, Diophantine geometry, and real algebraic geometry. I will present an overview of known results, and discuss existing algorithmic challenges and open problems.

This is joint work with James Worrell.
Contributions
**Abstract**—In this paper, we aim at developing a unique compression technique which "breaks the downward spiral" between compression ratio and data fidelity. The recently-developed Fuzzy Transform is exploited as a sensor data compressor, called Fuzzy Transform Compression (FTC). Based on contrasting FTC to other compressors, we design and implement a modified version of the FTC algorithm, referred to as FuzzyCAT–Fuzzy Compression: Adaptive Transform. FuzzyCAT adapts the transform parameters in accordance with the signal curvature, which could be inferred from the signal derivatives, to accomplish the optimal balance between compression ratio and precision. Generally, FuzzyCAT provides the users/apps with full control to prioritize either compression ratio or precision according to the significance. FuzzyCAT considerably outperforms the original FTC, whereas preserving its favorable qualities like periodicity and resilience to lost packets. Moreover, a full appraisal depicts the FuzzyCAT eclipses the LTC at compression ratios above 75. A series of experiments with a network of TelosB sensor nodes revealed that transmission costs of the FuzzyCAT algorithm is 96% less than that of the LTC at the expense of 10.28% increase of processing activities, which makes it an outstanding candidate for data compression in WSN.

**Keywords**—Wireless Sensor Networks; Energy Efficiency; Fuzzy Transform; Data Compression

**I. INTRODUCTION**

Wireless sensor networks (WSNs) have a wide range of potential applications to industry, science, transportation, civil infrastructure, and security. For instance, Industrial applications of WSNs are projected to extend by 55% within the five years, to nearly 24 million installed sensor points [1]. Energy efficiency plays a vital role in the WSNs wide-spreadening.

A rational methodology to mitigate the energy consumption problem could commence with identifying the major energy consumption sources to be tackled. A set of experiments with TelosB nodes has been executed to identify the dominant factor of energy consumption within each sensor node. Specifically, the CPU consumes much less current than the transceiver. This result is consistent with the energy model of the TelosB energy model listed in Table I. Declining the transceiver’s activities via data manipulation could significantly extend the lifetime.

Over the past decade, most research in WSNs has emphasized the data reduction significance for long-life networks. Data compression is classified into lossless and lossy approaches. The former has zero recovery error with relatively small compression ratios, which makes it suitable for applications requiring high precision like patients monitoring in health care. Lossy compression methods, on the other hand, incur recovery errors but achieve higher compression ratios. For applications that can tolerate some information loss, like environmental monitoring and other types of data logging, lossy compression techniques afford high savings in terms of power consumption at the minor cost of permissible reconstruction error. Moreover, lossy algorithms tend to be less complex than their lossless counterparts, hence are easier to implement on the computationally constrained motes.

In this work, we refine our proposed fuzzy compression algorithm (FTC) [2] for the sake of improving its accuracy. Additionally, we differentiate our novel fuzzy transform compression to the lightweight temporal compression technique [3]. In fact, this article deals with the latter setting, where data transmission dominates all other contributions to energy dissipation, such that techniques like sleep modes for transceivers or sensor nodes between sensing phases, efficient routing and topology control have only limited impact.

Our strategy to improve the FTC precision is to tracking the signal curvature. Thus, a novel version of the FTC, referred to as fuzzy compression adaptive transform (FuzzyCAT) is presented. Its core idea is to adapt the transform parameters to...
the signal’s curvature inferred from the time derivatives. The paper comprises a variety of simulations and real experiments with TelosB sensor nodes. These evaluations aim at comparing the performance of FTC, FuzzyCAT and LTC in terms of time/space complexity and energy consumption. For data recovery precision, the results show the superiority of FTC and FuzzyCAT over LTC algorithm for CRs above 50. Moreover, FuzzyCAT saves 96.07% of the radio energy consumption at the expense of consuming 10.28% more processing energy consumption over that of the LTC algorithm.

The remainder of the paper is organized as follows. Section II briefly formulates the accuracy conflict within lossy compression techniques. Section III summarizes the recent efforts in data compression for WSNs. Section IV discusses the idea behind the proposed FuzzyCAT approach. Moreover, plenty of performance evaluations, outlining the merits and flaws, are introduced. Finally, Section V concludes the paper and provides some suggestions for possible extensions.

II. PROBLEM FORMULATION

The sensor readings are forged as a time series \( X = \{x[1], x[2], \ldots\} \), where \( x \in \mathbb{R}^n \) is the ADC output due to an observed physical phenomenon. The term \( X[i : j] \) denotes a data window in the period \( i \leq n \leq j \), i.e., \( X[i : j] = \{x[i], x[i+1], \ldots, x[j]\} \). At the bases station, an approximated version of \( X \) is generated such that \( Y = \{y[1], y[2], \ldots\} \) where \( y \in \mathbb{R}^n \).

A rate here is defined as the average number of bits used to represent a subseries \( X[i : j] \). The rate distortion function \( R(D) \) is typically utilized to characterize the trade-offs between rate and the lossy compression distortion \( D \) [4]. The function \( R(D) \) specifies the lowest rate at which the output of a source can be encoded while keeping the distortion less than or equal to \( D \). Recall that the general form of the distortion is

\[
D = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} d(x_i, y_j) \times P(x_i) \times P(y_j | x_i) \tag{1}
\]

where \( d(x_i, y_j) := |y_j - x_i| \) is the Euclidean distance between these two sequences, \( P(x_i) \) is the source density, and \( P(y_j | x_i) \) is the conditional probability. Assuming a binary source with \( P(0) = p \), the rate distortion function is given in Eq. 2 in terms of the the probability \( p \) and the distortion \( D \). The function \( H(.) \) denotes the entropy as a measure of the average amount of information in the sequence. Clearly, minimum representation of the source data is feasible whenever the distortion \( D \) is minimized.

\[
R(D) = H(p) - H(D) \text{ for } D < \min\{p, 1 - p\} \tag{2}
\]

III. RELATED WORK

In this section, we briefly discuss the idea behind the Fuzzy compression technique (FTC) the lightweight temporal compression. Moreover, we present a comparison between them which motivated us to refine the fuzzy compressor.

A. Fuzzy Encoding

Fuzzy transform is defined as a fuzzy set mapper of a continuous/discrete function into an n-dimensional vector [5]. Assume a time series is confined into an interval \( [a, b] \) as a universe. This domain is fuzzy-partitioned by Fuzzy sets given by their membership functions.

Definition 1. Suppose uniformly distributed nodes \( x_1 \leq \ldots \leq x_n \) with \( \phi \) such that \( n \geq 2 \). The fuzzy sets \( A_{1}, \ldots, A_{n} \) are referred to as a uniform basic function whenever they conform to the following conditions for \( k = 1, \ldots, n \):

1) \( A_k : [a, b] \rightarrow [0, 1], A_k(x_k) = 1 \)
2) \( A_k = 0 \) if \( x \notin (x_{k-1}, x_{k+1}) \)
3) \( A_k \) is continuous over \( \phi \)
4) \( A_k \) rigorously increases on \( [x_{k-1}, x_k] \) and rigorously decreases on \( [x_k, x_{k+1}] \)
5) \( \sum_{k=1}^{n} A_k(x) = 1 \) \( \forall x \in [a, b] \)

Figure 1 depicts an example of a uniform triangular basic function with equidistant nodes given by Eq. 3. The red line delineates condition 5 where summation of any two vertical points should equal one. Generally, the shape of basic functions forges the approximating function. Hence, the F-transform is well-suited for dealing with linear and non-linear sensor readings.

![Fig. 1: Uniform triangular basic function](image_url)

\[
x_k = a + \frac{(b - a)(k - 1)}{(n - 1)} \tag{3}
\]

Strictly speaking, direct F-transform converts the original signal into an n-dimensional vector, where \( n \) corresponds to the number of membership functions applied. Inverse F-transform, on the other hand, approximates the original signal utilizing the Fuzzy vector. The F-transform is explicitly defined for discrete as well as continuous functions.

Definition 2. Assume a fuzzy partition of \( \phi \) be given by basic functions \( A_{1}, \ldots, A_{n} \subset \phi \) and \( n > 2 \). If a F-transformer has been triggered with a discrete function \( f : \phi \rightarrow \mathbb{R} \) known at nodes \( x_1, \ldots, x_n \) such that for each \( k = 1, \ldots, n \), there exists \( j = 1, \ldots, l \) : \( A_k(x_j) > 0 \). Then, the \( n \)-tuple of real numbers \( [F_1, \ldots, F_n] \) is given by

\[
F_k = \frac{\sum_{j=1}^{l} f(x_j)A_k(x_j)}{\sum_{j=1}^{l} A_k(x_j)} \tag{4}
\]

Fuzzy control theory is crucial for understanding the F-transform essence. Specifically, the direct F-transform resembles the defuzzification process (Center of gravity) through...
which linguistic variables (“low”, “medium”, “high”, etc.) are mapped onto real numbers. This implies that each vector element \( F_k \) constitutes the weighted average of the data points \( f(x_j) \in [x_{k-1}, x_{k+1}] \).

**Definition 3.** Suppose a fuzzy vector \( F_n[f] = [F_1, ..., F_n] \) w.r.t. \( A_1, ..., A_n \) has been stimulated to an inverse F-transformer. The recovered signal is given by

\[
f_{F,n}(x) = \sum_{k=1}^{n} F_k A_k(x)
\]

The basic functions characteristics such as their shape and length, devote a fine-grained control over the recovery process. Therefore, they have to be carefully designed to avoid imperfect transformation. The interested readers can find more properties and proofs in [5].

### B. Lightweight Temporal Compression

Targeting environmental applications such as temperature, humidity, and light sensing, the LTC algorithm [3] exploits the signal’s high temporal correlation to approximate it by a sequence of line segments. The information loss is controlled by a user-set error margin: whenever the approximating line deviates from the next data point by more than the error margin, the current line parameters are sent and a new approximation is started.

In the context described above, our work lies in the realm of lossy compression at individual source nodes. Therefore, to give a fair point of reference, we will compare our algorithm with that of the LTC, as the original and the more well-known version of data linear approximation.

To compare FTC performance against that of LTC, we ran a series of experiments on authentic sensor data acquired by the Berkeley Lab in 2005 [6]. In particular, the signal used for the assessment came from the light sensor of node #50. Figure 2 represents a segment of time series light intensity data in its original form, as well as two recovered time series utilizing LTC and FTC decompressors. As a metrics for performance evaluation, we used the compression ratio (CR) and the normalized root mean square error (RMSE).

\[
CR = \frac{\text{Uncompressed size}}{\text{Compressed size}}
\]

\[
RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (y_i - y_{i,\text{reconstructed}})^2}
\]

\[
\text{normalized RMSE} = \frac{\text{RMSE}}{y_{\text{max}} - y_{\text{min}}} \times 100\%
\]

Despite both algorithms exhibit similar performance. However, the FTC in its original form is completely irrespective of the properties of the data being compressed. This is sometimes disadvantageous. Implementing an adaptive approach to data processing is therefore a crucial improvement for the FTC algorithm.

Algorithm 1 depicts the mechanism by which signal curvature is detected. The number \( \omega \) of data points to be compressed, the base number \( n \) of basic functions to be applied to the Fuzzy universe, as well as the number \( e \) of extra basic functions to be applied per half period, are set by the user such that (1) \( (n - 1) \omega \) and (2) \( (e + 1) \frac{\pi}{n \omega} \). The data is acquired through iterating over half periods of a basic function. Throughout the process, the program maintains a meta array where each cell is set if the corresponding half period requires higher resolution. For each data point, with an obvious exception of the first two, the second derivative \( \frac{d^2}{dt^2} f(t) \) is computed. Knowing the high noise level seen in sensed environmental data, it is important to only increase the resolution of the transform if the fluctuations detected are significant. To ensure this, two stages of filtering are applied. In the first stage, the current \( \frac{d^2}{dt^2} f(t) \) is compared to the derivative threshold \( T_{deriv} \) set by the
If one were to graph the resulting basic functions over the whole time window, then one would see something like the graph given in Figure 4. The sample signal is shown on top, and the fuzzy sets constructed by FuzzyCAT for that signal are displayed on the bottom. On the half periods where the signal is smooth, the regular membership functions are applied. In the half period where fluctuations were detected, narrower basic functions are applied (in blue). Note that to fulfill the requirement (5) of Definition 1, the basic functions adjacent to the high-resolution half period (in red) are asymmetric. They represent so-called hybrids because they are constructed using part of a regular and an additional basic function. As a result, the area around the points with high $\frac{d^2f(t)}{dt^2}$ is transformed using smaller basic functions, thus ensuring higher precision.

Figure 5 presents an example of comparison between the regular FTC and FuzzyCAT performance on a segment of the temperature signal from the Berkely lab dataset. Both algorithms aimed to compress the 1000 data points into 26 coefficients, while FuzzyCAT was set to add three additional basic functions per half period when needed. The scaled pink line, representing the difference between the signal reconstructed by the regular FTC and FuzzyCAT, reveals that the algorithms yielded identical results on most of the segment, only deviating on the intervals with high fluctuations. The FTC yields compression ratio of 38.46, with normalized RMSE of 8.72%. The adaptive transform added 9 extra membership functions, decreasing the compression ratio to 28.57 and bringing the normalized RMSE down to 4.22%. Adding extra membership functions cut the RMSE by more than half - a 52% decrease, while the resulting compression ratio was only 25% percent smaller than the original. Thus, FuzzyCAT exhibits a compelling advantage over the regular F-transform.

Figure 6 shows the results of the comparison. Note that depending on the error margin, LTC can yield different reconstruction errors with the same compression ratio. LTC performs best, when CR is under 50, after which the FuzzyCAT is likely to perform just as well. For a CR above 75, FuzzyCAT and FTC outperform the LTC technique.

To evaluate how FuzzyCAT and LTC affect the energy consumption, we ran a series of experiments using TelosB sensor nodes (CM5000 MSP) with Contiki OS. The setup
involved a network under ConikiMAC radio duty cycling protocol with the *unicast* communication primitive in the Rime stack. A network of three nodes: a LTC node, a FuzzyCAT node, and a sink node was established. The run-time power consumption was estimated utilizing the *Energest* module in the Contiki OS. For the fairness of the experiment, the parameters of each algorithm were set such that both resulted in the same normalized RMSE. The LTC node sent, on average, 53 packets whereas the FuzzyCAT transmitted solely 11 packets for the same data received at the sink. Figure 7a delineates the power consumed via broadcasting the LTC and the FuzzyCAT vectors. In fact, the FuzzyCat consumes 96.07% less energy than the LTC for a fixed throughput. This significant gain comes at the expense of 10.28% increase in the processing tasks as shown in Fig. 7b.

V. C O N C L U S I O N

In this paper, we presented a novel lossy compression algorithm for WSN called (FuzzyCAT), designed and implemented in the C programming language for Contiki OS. Testing the algorithm against the well-known LTC using a network of TelosB sensor nodes revealed that FuzzyCAT consumes 96.07% less transmission energy than LTC for a fixed throughput, which is the dominant source of power consumption in WSN.

Thus, despite the tongue-in-cheek name, FuzzyCAT is a very competitive candidate as a WSN compressor. Besides being more energy efficient and less computationally complex than LTC, it is characterized by periodicity, a property that increases the resilience to lost packets and makes the algorithm compatible with scheduling protocols. On the other hand, FuzzyCAT possesses such negative qualities as high latency, potentially overusing the limited storage capacity of the mote and the packets, and complex parameters requiring careful optimization. However, we believe that FuzzyCAT is a promising technique for data compression in WSN targeting such applications as environmental monitoring and other data logging.

As opportunities for future work, we consider setting up predictors for mitigating the effect of long delay inherent in the compression process.

REFERENCES


A Nivat Theorem for Weighted Picture Automata and Weighted MSO Logic

Parvaneh Babari (babari@informatik.uni-leipzig.de)

Institut für Informatik, Universität Leipzig, 04109 Leipzig, Germany

The theory of picture languages as a generalization of formal string languages was motivated by problems arising from image processing and pattern recognition [20, 32], and also plays a role in the theory of cellular automata and other devices of parallel computing [29, 35]. In the nineties, the family of recognizable picture languages was defined and characterized by many different devices [21, 23]. Several research groups obtained a description of recognizable picture languages in terms of automata, sets of tiles, rational operations, and existential monadic second-order logic [22, 24, 25, 28]. Bozapalidis and Grammatikopoulou introduced the interesting model of weighted (quadrapolic) picture automata whose transitions carry weights taken as elements from a given commutative semiring [4]. The behavior of such a picture automaton is a picture series which maps pictures over an arbitrary alphabet to elements of the semiring. In 2006, Fichtner provided a notion of a weighted MSO logic over pictures [17, 18, 19]. She proved that for commutative semirings, the class of picture series defined by sentences of the weighted logics coincides with those computed by weighted picture automata [17].

In this paper we define picture valuation monoids as the abstract model for the weight structures and we introduce weighted two-dimensional on-line tessellation automata (W2OTA) taking weights from picture valuation monoids. By this, we can model several application examples, e.g., the average light of picture (interpreting the alphabet as different levels of light) which can not be modelled with commutative semirings. Weighted automata over words computing objectives like the average cost were introduced recently by Chatterjee, Doyen, and Henzinger [5, 6, 7, 8].

As our first main result, we prove a Nivat-like theorem for recognizable picture series, i.e., for the behaviors of W2OTA. Nivat’s Theorem is a fundamental characterization of rational transductions and provides a connection between rational transductions and rational languages; see [10] for a version of this result for semiring-weighted automata on words. Recently, Droste and Perevoshchikov [12] proved a Nivat-like theorem for recognizable quantitative timed languages. Here, we will derive such a result for recognizable picture series. We show that recognizable picture series can be obtained precisely as projections of particularly simple unambiguously recognizable series restricted to unambiguously recognizable picture languages. In addition, we show that if the underlying picture valuation monoid is idempotent, then we do not need unambiguity of the underlying picture language.

*Joint work with Manfred Droste
†Supported by DFG Graduiertenkolleg 1763 (QuantLA)
In the second part of this paper we define a new weighted MSO logic which can model average density of pictures. The weighted MSO logic used here is a combination of the ideas from [3], [11], [12] and [17]. In [17], disjunction and existential quantification were interpreted by the sum, and the semantics of both conjunction and universal quantification were defined by the product operation of the semiring. In this paper, using picture valuation monoids as the abstract model, the semantics of universal quantification will be interpreted by a picture valuation function, which for example provides the average value of light of a picture.

Our second main result states that the weighted automata device of W2OTA and a fragment of weighted MSO logic are expressively equivalent. To reach this result, we define a suitable fragment of our logic in which the application of universal first order (FO) quantification is restricted to almost boolean FO formulas, and the application of conjunction is restricted to either almost boolean FO formulas or boolean FO formulas. In addition, we restrict the use of constants in the formula by allowing their occurrence only in the scope of an FO universal quantifier. This enables us to derive our second main result for arbitrary product picture valuation monoids, not requiring regularity as in [11]. Also, our results differ from the ones in [17] which required commutative semirings as weight structure.

We would like to mention that our results do not need distributivity of multiplication over addition or commutativity or even associativity of multiplication, while considering a commutative semiring as the weight structure was previously an essential assumption in the weighted picture automata theory.

References


A Nivat Theorem for Weighted Picture Automata and Weighted MSO Logics


Deciding Monadic Second Order Logic over $\omega$-words
by Specialized Finite Automata

Stephan Barth (stephan.barth@ifi.lmu.de)

Ludwig-Maximilians-Universität München, Germany
Research Training Group 1480: Programm- und Modell-Analyse (PUMA)

Introduction

Several different automata models can describe all $\omega$-regular languages. The most commonly used models for that are Büchi, parity, Rabin, Streett and Muller automata. We present deeper insights and further enhancements to a lesser known model. This model was chosen and the enhancements developed with a specific goal: Decide monadic second order logic (MSO) over infinite words more efficiently.

MSO over various structures is of interest in different applications, mostly in formal verification. Due to its inherent high complexity, most solvers are designed to work only for subsets of MSO. The most notable full decider is MONA, which decides MSO formulae over finite words and trees.

The MONA team identified efficient minimization as one of the crucial properties an automaton model has to support to decide MSO efficiently[2]. The classical models for $\omega$-languages do not support that. The other central optimizations MONA used are compatible with all the classical $\omega$-automata models (these are BDD-compression of the alphabet and three-valued logic; our here presented model is compatible with these as well).

To obtain a suitable automaton model, we further studied a representation of $\omega$-regular languages by regular languages[1]. We succeeded in developing an algorithm for homomorphisms in this representation, which is crucial for deciding MSO. For even higher efficiency, we contribute a new automaton model for regular languages, that is more succinct than finite automata, especially for this kind of regular languages.

Representation of $\omega$-regular languages by regular languages

The starting point is the following method for representing $\omega$-regular languages by regular ones: for given $\omega$-regular languages $L$, $L_\omega := \{u\$v \mid uv^\omega \in L\}$ is regular.

We call $L_\omega$ the loop language of $L$, an automaton for $L_\omega$ loop automaton, and $L$-$X$ the loop automaton model that results from using the automaton model $X$ for loop languages.

Transformations between nondeterministic Büchi automata (NBA) accepting $L$ and deterministic finite automata (DFA) accepting $L_\omega$ were presented in 1994[1].
Note, that $L_\$ and thus the minimal DFA are uniquely determined, hence the known efficient minimization procedure for DFA work for L-DFA as well.

It is thus natural to base a decision procedure for MSO on loop automata. However, in doing so one faces the obstacle that homomorphism has to be implemented on the level of loop automata.

Indeed this obstacle seems to have prevented other authors from following this path.

It is precisely this gap we close in this paper.

### Homomorphism Closure of Loop automata

The crucial idea to perform homomorphisms is the use of two-way automata[3] as intermediate device.

Given an homomorphism $f$, it happens often, that $f(L_\$) \neq (f(L))_\$, hence applying the homomorphisms directly on the regular language does not yield the correct result.

Instead $\{uv \mid \exists i,j. uv^i v^j \in f(L_\$)\} = (f(L))_\$. A nondeterministic two-way-automaton with linear many states for this language can be constructed out of $L$, transformations from nondeterministic two-way-automata to DFA are known for a long time.

Hence, homomorphisms on loop automata can be computed, and with that loop automata can be used for deciding MSO.

### Further optimization of loop automata

This method works fine in practice, as for example the comparisons in Table 2 show, but we can do better. L-DFA are often bigger than DPA or NBA for the same language. We identified one of the main sources for their bigger state space: Loop automata tend to contain a lot of strongly connected components (SCC), which are identical but in the finality of their states.

As a countermeasure, we developed a new automata model for regular languages, coarser merging finite automata (CMFA), which can often merge these different SCC into a single one. CMFA have a strongly restricted stack. Due to these restrictions, they still only recognize regular languages. As a result from these restrictions, CMFA allow for states in Nerode relation to be merged just like in DFA. But because of the stack, they admit further state merges within a new equivalence relation, the mutual right derivative (MRD) relation, which is coarser than the Nerode relation: Two states are in MRD relation, when their languages are mutual right derivatives by some words $u$ and $v$. States in MRD relation can be merged in CMFA, whenever they are in different SCC. The necessary $u$ and $v$ have to be stored in the CMFA at the corresponding transition.

Hence, CMFA offer more minimization opportunities than DFA.

CMFA are defined like a DFA but the transition function is changed to

$$\delta : Q \times \Sigma \rightarrow Q \times \Sigma^* \times \Sigma^*$$

when $Q$ is the state set and $\Sigma$ the alphabet.

$\delta$ must hold the property that for every tuple $\delta(p, \_ ) = (q, w, r)$ if $p$ and $q$ are in the same SCC of the automaton, then $w = r = \varepsilon$.

$(w, r)$ is a witness for $q$ and some removed state to be in MRD relation and can be used to reconstruct this removed state.
Deciding Monadic Second Order Logic over omega-words by Specialized Finite Automata

| $\omega$-regular expression | $(a|b)^*b^\omega$ | $(a|b)^*(ab)^\omega$ |
|-----------------------------|------------------|---------------------|
| **B"uchi automaton**        | ![Buchi Automaton](image1) | ![Buchi Automaton](image2) |
| **Loop regular expression** | $(a|b)^*b^+$ | $(a|b)^*$($((ab)^+|(ba)^+)$) |
| **Loop DFA**                | ![Loop DFA](image3) | ![Loop DFA](image4) |
| **Loop CMFA**               | ![Loop CMFA](image5) | ![Loop CMFA](image6) |

Table 1: Two examples of languages for comparing the $\omega$-regular language and its corresponding loop language

Table 1 displays the representation of two example languages in the different models.

Unfortunately, full minimization of CMFA is not very efficient. Nevertheless, there is an efficient heuristic which guarantees a partial minimization to at most the size of DFA, and in practice often leads to minimal CMFA, or at least CMFA which are not much bigger than the minimal ones.

While CMFA are up to exponentially more succinct than DFA, the relevant algorithms for deciding MSO are still applicable in an efficient manner.

**Complexity**

It was already known that NBA of size $n$ can be transformed into L-DFA of size $2^{O(n^2)}$ and L-DFA of size $n$ into NBA of size $O(n^5)$[1].

This of course can be used to performe homomorphisms on L-DFA in $2^{O(n^{10})}$ states, as homomorphisms on NBA keep the state count. Our new algorithm needs at most $2^{O(n^2)}$ states for that. Complementation needs $O(n)$ states. Union and intersection are performed on L-DFA precisely by performing them on the DFA. This leads to the same $O(n \cdot m)$ complexity as DFA have.

For L-CMFA complementation also needs at most linear many states, Homomorphism, union, and intersection might need more states in L-CMFA regarding
the size of the input automata and the precise sizes are unknown, yet. Nevertheless the size is always bound by the size of the corresponding L-DFA.

**Experimental Evaluation**

For benchmark purposes, we compared the efficiency for deciding MSO between this new loop CMFA (L-CMFA) model, loop DFA (L-DFA) and a more classical approach utilizing NBA and deterministic parity automata (DPA) together with state reduction heuristics.

As there are some widely used minimization heuristics for NBA and DPA, we also applied a minimization heuristic for them. In the DPA, we check for every pair of states, whether the language of the DPA stays the same, if the states are merged. This heuristic subsumes quite some widely used heuristics, but is not frequently used as such, as it is too slow; nevertheless, it is still too weak for deciding MSO, at least in comparison to loop automata.

The L-CMFA approach indeed turns out to be a lot more efficient for many formulae than the NBA/DPA-approach. In these cases, the CMFA compression contributes very little to the efficiency in comparison to DFA. However, for some formulae the NBA/DPA-approach is slightly more efficient than the L-CMFA approach. In these cases, the CMFA compression tends to contribute more to the efficiency of the procedure.

<table>
<thead>
<tr>
<th>Formula</th>
<th>NBA/DPA</th>
<th>L-DFA</th>
<th>L-CMFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>after((X, Y) := \forall x. (x \in X \Rightarrow \exists y. (y &gt; x \land y \in Y)))</td>
<td>6/3</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>fair((X, Y) := \text{after}(X, Y) \land \text{after}(Y, X))</td>
<td>42/27</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>(\forall X. (\text{fair}(X, Y) \Rightarrow \text{fair}(Y, Z)))</td>
<td>(14377)/(6131)</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>inone((i, j, U, V, W) := (j \in U \Rightarrow i \in V \lor i \in W))</td>
<td>20/5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(\forall i \forall j. (\text{suc}(i, j) \Rightarrow \text{inone}(i, j, V, W, Z) \land \text{inone}(i, j, W, X, V) \land \text{inone}(i, j, Y, Z, X) \land \text{inone}(i, j, Z, V, Y)))</td>
<td>(t/o)/</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>offset((X, Y) := \forall i \forall j. (\text{suc}(i, j) \land j \in Y \Rightarrow i \in X))</td>
<td>4/3</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>offset((X, Y) \land \text{offset}(Y, Z) \land \text{offset}(Z, X))</td>
<td>49/40</td>
<td>107</td>
<td>61</td>
</tr>
<tr>
<td>offset((V, W) \land \text{offset}(W, X) \land \text{offset}(X, Y) \land \text{offset}(Y, Z) \land \text{offset}(Z, V))</td>
<td>97/(444)</td>
<td>2331</td>
<td>583</td>
</tr>
</tbody>
</table>

Table 2: State count of automata from MSO formulae

Most of the tested formulae fall into one of three classes:

- The formula is small, hence both approaches are equivalently fast.

1 Timeout in minimization of DPA. With weaker minimization, it ended up in this result.

2 Computation did not finish in over a day. When it was stopped, it was in the process of computing a DPA out of an NBA and had already over 190000 states.

3 Minimization of parity automaton did not finish in a day.
The loop automaton is much smaller than the automaton from the NBA/DPA approach, the CMFA compression does not make a big difference.

The NBA/DPA approach is more efficient than the L-DFA approach, but L-CMFA regain most of the efficiency.

Table 2 shows some typical results of the experimental evaluation. It contains at least one example for each of the three classes. Efficiency here is measured in state count of the automata, because this only depends on the adequateness of the automata models, but not on the quality of the implementation.

Conclusion

For loop automata we now know all necessary algorithms to decide MSO. Hence, we now have an automaton model for $\omega$-regular languages, that is suitable for deciding MSO and allows for efficient minimization at the same time. Along with its applicability for MSO, L-CMFA might offer further theoretical and practical enhancements in the field of $\omega$-languages.

On the experimental side, the first benchmarks hint that loop automata, especially L-CMFA, are indeed superior to classical automata for $\omega$-regular languages in efficiently deciding MSO. A full implementation of MSO over $\omega$-words utilizing L-CMFA and integration of other optimizations MONA uses, is in development.

References


From sweeping word transducers to one-way word transducers*

Félix Baschenis, (felix.baschenis@labri.fr)
LaBRI, Université de Bordeaux, France

1 Introduction

Automata theory provides many different computational models that can deal with the same classes of problems. One can extend automata by non-determinism, alternation, or a two-way head and obtain the same class of languages in the case of finite-state automata. However it can be difficult to express the memory cost of the transformation from one model to another. Moreover, several properties do not extend from automata when we add outputs, i.e., when we consider transducers. For instance, non-deterministic two-way transducers (2NFT for short) are more powerful than their one-way counterpart (NFT for short). As an example, the transduction mapping the input to its mirror can be realized by a 2NFT, but not by an NFT. We are interested here in the following question: given a 2NFT, is it possible to find an equivalent NFT? We consider here only functional NFTs, that is NFTs that produce at most one output for each input word.

The above problem was addressed by one of my supervisors in [1]. They proved that the above question is decidable, and gave a non-elementary procedure to build an equivalent NFT when possible. Their approach analyzes runs of the 2NFT. They identify in each run some elementary two-way motions on the input tape, called Z-motions. Such motions have a zigzag shape between two positions $i$ and $j$: the input head moves from $i$ to $j$, then from $j$ to $i$, and finally back from $i$ to $j$. They show that if the given 2NFT is equivalent to an NFT, then it is possible to eliminate Z-motions one by one until an equivalent NFT is obtained. Since each step involves an exponential blow-up in the number of states, the overall procedure given in [1] has non-elementary complexity.

Our goal is to extend the ideas of [1] in order to provide a direct construction of an equivalent NFT that avoids the non-elementary complexity blow-up.

We provide here such a direct construction for a particular class of 2NFTs, namely sweeping transducers (SNFT for short). An SNFT is a 2NFT whose head reversals on the input tape can only occur at the left or at the right border of the tape. Our construction of NFT from a given SNFT has doubly exponential complexity.

*Supported by Anca Muscholl, Olivier Gauwin and Gabriele Puppis
2 A simple example

We explain some ideas behind our construction by considering first sweeping automata with only one reversal, i.e. back-and-forth transducers that start reading the input leftmost, go rightmost and then back to the left border.

The well-known combinatorial result below [2] will be used a lot in our proofs. An integer \( p \in \mathbb{N} \) is called period of a word \( w = a_1 \cdots a_n \) if \( a_i = a_{i+p} \), for every \( 1 \leq i \leq n-p \). The primitive root of a word \( x \) is denoted \( \mu(x) \). For \( u, v \in \Sigma^* \) we write \( u \sim v \) if the primitive roots of \( u \) and \( v \) are conjugated, i.e., \( \mu(u) = xy \) and \( \mu(v) = yx \) for some words \( x, y \).

**Theorem 2.1 (Fine and Wilf)** If \( w \) is a word of periods \( \pi_1 \) and \( \pi_2 \) and \( |w| > \pi_1 + \pi_2 - \gcd(\pi_1, \pi_2) \), then \( w \) has also period \( \gcd(\pi_1, \pi_2) \).

An equivalent formulation of Fine and Wilf’s theorem states that if \( u^\omega \) and \( v^\omega \) have a common factor of length greater than \( |u| + |v| - \gcd(|u|, |v|) \), then \( u \sim v \).

We define the crossing sequence of a run of a two-way automaton at a position \( i \), as the sequence of states the transducer takes at each successive pass at position \( i \), together with the letter of the input at position \( i \). When two positions \( i \) and \( j \) have the same crossing sequences, they form what we call a loop: if we pump the input between those positions, the part of the run between \( i \) and \( j \) is equally duplicated. We write \( M = |Q|^2 \times \mathbb{P} \times |\Sigma| \) where \( \Sigma \) is the alphabet of the back-and-forth 2NFT, and \( p \) is the maximal size of an output of a transition. So \( M \) is the maximal size of an output produced by a back-and-forth 2NFT in a run without loops. When we consider a two-way transducer, we write \( \text{out}_k(i, j) \) for the output produced on the \( k \)-th pass, between the position \( i \) and \( j \) (the order between \( i \) and \( j \) is determined by \( k \)). We also use the notation \( \text{out}(i, k, (j, \ell)) \) for the output produced by the run of the transducer between position \( i \) on the \( k \)-th pass and position \( j \) on the \( \ell \)-th pass. A loop \( L \) is denoted by a pair \( (i, j) \) of input positions \( i < j \). For two loops \( L = (i, j), L' = (i', j') \) we write \( L < L' \) if \( j < j' \).

The following propositions state some sufficient conditions under which parts of the output of the 2NFT are periodic, with small period. The periodicity is consequence of Fine & Wilf’s theorem, and we obtain it by considering word equations related to loops in the run. Such periodic outputs can be produced by an NFT, once the period is guessed.

**Proposition 2.2** Let \( \mathcal{A} \) be a back-and-forth 2NFT that is equivalent to some NFT. We consider a run \( \rho \) of \( \mathcal{A} \). If \( L = (i, j) \) is a loop of \( \rho \) such that \( \text{out}_2(i, j) \neq \varepsilon \) and \( \text{out}_1(i, j) \neq \varepsilon \), then \( \text{out}((i, 1), (i, 2)) \) has period at most \( M \).

**Proposition 2.3** Let \( \mathcal{A} \) be a back-and-forth 2NFT that is equivalent to some NFT. We consider a run \( \rho \) of \( \mathcal{A} \). Assume that \( L_1 = (i_1, j_1) \) and \( L_2 = (i_2, j_2) \) are two loops of \( \rho \) such that \( L_2 \geq L_1 \), \( \text{out}_2(i_1, j_1) \neq \varepsilon \), \( \text{out}_1(i_1, j_1) \neq \varepsilon \) and \( \text{out}_2(i_2, j_2) = \varepsilon \). Then \( \text{out}((i_2, 1), (i_1, 2)) \) has period at most \( M \).

**Proposition 2.4** Let \( \mathcal{A} \) be a back-and-forth 2NFT that is equivalent to some NFT. We consider a run \( \rho \) of \( \mathcal{A} \). Assume that there exist two loops \( L_1 = (i_1, j_1) \) and \( L_2 = (i_2, j_2) \) of \( \rho \) with \( L_2 \geq L_1 \) such that \( \text{out}_2(i_1, j_1) \neq \varepsilon \), \( \text{out}_1(i_1, j_1) = \varepsilon \), \( \text{out}_1(i_2, j_2) = \varepsilon \) and \( \text{out}_2(i_2, j_2) \neq \varepsilon \). Then \( \text{out}((j_2, 2), (i_1, 2)) \) has period at most \( M \).
Proposition 2.5 Let $A$ be a back-and-forth 2NFT that is equivalent to some NFT. We consider a run $\rho$ of $A$. Assume that there exist two loops $L_1 = (i_1, j_1)$ and $L_2 = (i_2, j_2)$ in $\rho$ with $L_2 \geq L_1$ and such that $\text{out}_2(i_1, j_1) \neq \varepsilon$, $\text{out}_1(i_1, j_1) = \varepsilon$, $\text{out}_1(i_2, j_2) \neq \varepsilon$ and $\text{out}_2(i_2, j_2) \neq \varepsilon$. Then $\text{out}(\langle i_2, 1 \rangle, \langle i_1, 2 \rangle)$ has period at most $M$.

Theorem 2.6 Let $A$ be a back-and-forth 2NFT. One can construct an exponential size NFT $B$ such that

1. $B \subseteq A$, and

2. $\text{dom}(B) = \text{dom}(A)$ if $A$ is equivalent to some NFT.

Proof. Consider a run $r$ of $A$ on input $U$ with output $V$. There are two main cases. The first one is when there is no loop $L$ of $A$ with $\text{out}_2(L) \neq \varepsilon$. In this case $V = V_1V_2$, and the output $V_2$ from right-to-left is small. The NFT $B$ guesses $r$, outputs $V_1$ and keeps $V_2$ in memory for the end.

The second case is where there is some loop $L$ with $\text{out}_2(L) \neq \varepsilon$. We consider the leftmost such loop $L = (i, j)$. Three subcases:

1. Assume that $\text{out}_1(L) \neq \varepsilon$. Then, by Proposition 2.2, the output $Y$ from position $\langle i, 1 \rangle$ to position $\langle i, 2 \rangle$ is periodic with period at most $M$. The entire output $V$ can be written as $V = XYZ$, with $|Z| < M$. The NFT $B$ outputs $X$ (guessing where it ends) and keeps in memory $Z$ for the end. It guesses then the period word of $Y$ and outputs $Y$ accordingly.

2. Assume that $\text{out}_1(L) = \varepsilon$ and there is no loop $L' > L$ such that $\text{out}_1(L') \neq \varepsilon$. If there is some loop $L' > L$ such that $\text{out}_2(L') \neq \varepsilon$ then we take the rightmost one $L' = (i', j')$. By Proposition 2.4, the output $Y$ from position $\langle j', 2 \rangle$ to position $\langle i, 2 \rangle$ is periodic with period at most $M$. The output $V$ can be written as $V = XXXYZ$ with $|X'| < 2M$, $|Z| < M$, with $X$ the output up to position $\langle i', 1 \rangle$. The NFT $B$ outputs $X$ (storing $Z$), guesses $X'$ and outputs it. Then it produces $Y$ (checking at the same time the guess of $X'$), and finally reproduces $Z$ that it has stored.

3. Assume that $\text{out}_1(L) = \varepsilon$ and there is some loop $L' > L$ such that $\text{out}_1(L') \neq \varepsilon$. We consider the first such loop $L' = (i', j')$. Then, by Proposition 2.3 or 2.5 (depending on the emptiness of $\text{out}_2(L')$), the output $Y$ from position $\langle i, 1 \rangle$ to position $\langle i, 2 \rangle$ is periodic with period at most $M$. The output $V$ can be written as $V = XXXXYZ$, where $|Z| < M$, $X$ is the output up to position $\langle j, 1 \rangle$ and $X'$ up to $\langle i', 1 \rangle$, thus $|X'| < M$. The NFT $B$ outputs $X$ (storing $Z$) then guesses and outputs $X'$ immediately. From position $\langle i, 1 \rangle$ on, $B$ produces $Y$.

3 Some ideas about the general proof

3.1 Statement of the theorem

The difference between a general sweeping transducer, and a back-and-forth 2NFT is that loops of the sweeping automaton can have non-empty outputs on several levels. Thus,
The parts of the run that produce periodic outputs can expand on several levels too. This yields the notion of a block decomposition, namely a decomposition of the run in different blocks producing periodic outputs, blocks ordered from left to right on the input tape, and from the lower levels to the higher ones of the run. Such a decomposition is described in Figure 1. We say that a decomposition is suitable if it satisfies some properties on the size of the outputs outside each block: the upper leftmost part and the lower rightmost part of the run must produce bounded outputs. Our main result can now be stated:

**Theorem 3.1** Let $\mathcal{A}$ be an SNFT. Then $\mathcal{A}$ is equivalent to some NFT if and only if each accepting run has a suitable decomposition. Moreover, we can construct a doubly exponential size NFT $\mathcal{B}$ such that:

1. $\mathcal{B} \subseteq \mathcal{A}$, and
2. $\text{dom}(\mathcal{B}) = \text{dom}(\mathcal{A})$ if and only if $\mathcal{A}$ is equivalent to some NFT.

### 3.2 The decomposition

Here we describe the ideas behind the proof of the left-to-right implication of the previous theorem. The blocks are obtained by considering loops as in the back-and-forth case, the only difference if that we should look for loops on several levels of the run. When a pair of loops is found such that the lowest non-empty output of the rightmost loop is lower than the upper non-empty output of the leftmost loop, we identify a block with periodic output. The equations induced by the two loops, in the same way as the back-and-forth case, are more complex, because the number of words iterated is bigger, but they behave in the same way as the equations used in the previous section.

### 3.3 Construction of the NFT

We describe here how we build a NFT that simulates the sweeping automata, guessing some suitable block decomposition of the run and producing the output according to this decomposition. So this corresponds exactly to the right-to-left implication of Theorem 3.1.

The simulation of the run of the given SNFT is done in two separate modes, and can go from one to another by making non-deterministic choices.
The first mode of computation occurs when the head of the input tape is not inside a block. In this case, it simulates the output of the SNFT on the current level (the words $u_i$ in Figure 1), and simultaneously checks the guesses made by the NFT.

In the second mode, the transducer assumes that it is in a block, whose length will be determined by the non-deterministic choice when to go back to the first mode. The NFT produces outputs in a periodic pattern, using the guessed primitive roots. At the beginning of the block the NFT guesses the number of periods outputted on the right and the left of the block, and outputs the amount guessed. Then, it continues its simulation and outputs as many letters as there are on the outputs produced by the SNFT at all levels of the current block. When reaching the end of a block, the NFT has progressed in its production of the output until the rightmost, uppermost corner of the block, and can go on.

Let us consider the example of $u_2$ and $B_2$ of the decomposition in figure 1, to illustrate all the checks made by the NFT.

When we simulated the output of $B_1$ we also guessed (and outputted) the bounded output to the right of $B_1$ on levels 1 and 2. These guesses must be checked, so while we output $u_2$ we check those guesses. Since $u_2$ is not part of a block we also check that the output above $u_2$ is bounded. We need to remember this output: the parts on levels 4, \ldots, 7 will be outputted together with $B_2$, whereas levels 8, \ldots, 11 will be stored together with what we remembered above $u_1$ and $B_1$. Note that we only remember words up to length $M$.

Let us consider $B_2$. We need to check that the output from the beginning to the end of $B_2$ is of the form $t^*t'$, for some word $t$ of length at most $M$, i.e. the fact that $B_2$ is indeed a block. When we output in this mode we use the stored outputs on levels 4, \ldots, 7 and guess the outputs right of $B_2$ on levels 3, \ldots, 6, that will also need to be checked.

4 Conclusion

This work had two objectives. First, to provide an efficient way to simulate a sweeping transducer by an NFT. Moreover, the NFT built does more than being equivalent when the SNFT is NFT-definable. In some way, even when the SNFT is not NFT-definable, the NFT that we build computes the transduction of all the words that can be outputted by an NFT. Then, to decide if a SNFT is NFT-definable, one needs to check the equivalence between the two automata underlying the SNFT and the NFT that we have built.

The second goal of this work is the possibility of lifting the technique to the general model of 2NFTs. The decomposition that we obtained cannot be directly used in this case, but we think that a similar decomposition of the run of the transducer can be done in the general case.

References


Introduction

Most approaches to the synthesis of controllers in reactive systems, for instance [1, 5, 4, 2], involve synthesizing transition systems such as Mealy or Moore automata. Unfortunately, the resulting transition systems can be very large. For example, for certain specifications in linear temporal logic (LTL), the size of the smallest transition system satisfying the specification is doubly exponential in the length of the formula [6].

The subject of this presentation is the synthesis of controllers represented in a more succinct way: We consider structured reactive programs over a finite set $B$ of Boolean variables, building on work of Madhusudan [3]. The syntax of structured programs is defined by the following grammar, where $b$ stands for a variable in $B$ and $\vec{b}$ for a vector of variables:

\[
\begin{align*}
    \langle \text{exp} \rangle &::= \text{true} \mid \text{false} \mid b \mid \langle \text{exp} \rangle \land \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \lor \langle \text{exp} \rangle \mid \neg\langle \text{exp} \rangle \\
    \langle \text{prog} \rangle &::= b := \langle \text{exp} \rangle \mid \text{input } \vec{b} \mid \text{output } \vec{b} \mid \langle \text{prog} \rangle;\langle \text{prog} \rangle \mid \text{if } \langle \text{exp} \rangle \text{ then } \langle \text{prog} \rangle \text{ else } \langle \text{prog} \rangle \mid \text{while } \langle \text{exp} \rangle \text{ do } \langle \text{prog} \rangle
\end{align*}
\]

Intuitively, "input $\vec{b}$" reads an input symbol (i.e., a bitvector) and stores it in the variables $\vec{b}$. Conversely, "output $\vec{b}$" writes the output symbol consisting of the current values of the variables $\vec{b}$. (The length of the input and output bitvectors is fixed.) Such programs can be viewed as a trees, as shown in figure 1.

A program is called reactive if it never terminates and each of its computations reads an infinite sequence of input symbols and writes an infinite sequence of output symbols.
2 Synthesis of Structured Reactive Programs

For a given $\omega$-regular specification $R$, representing the permissible input/output sequences, and a given finite set $B$ of Boolean variables, the synthesis problem asks to construct a structured reactive program over $B$ satisfying the specification (if such a program exists). We assume that the specification is provided in the form of a nondeterministic Büchi automaton $A_R$ that recognizes the complement of the specification.

Madhusudan [3] proposes a solution to this problem based on two-way alternating $\omega$-automata on finite trees (representing programs), which can then be transformed into equivalent nondeterministic tree automata (NTAs).

We provide a more elementary approach by directly constructing a deterministic bottom-up tree automaton (DTA) recognizing the set of correct programs, without a detour via more intricate types of automata. The DTA inductively computes a representation of the behavior of a given program in the form of so-called co-execution signatures. A co-execution is a pair consisting of a computation of the program and a corresponding run of the specification automaton $A_R$. A co-execution signature for a given program and a given specification automaton captures the essential information about their possible co-executions. In particular, such a signature suffices to determine whether there exists a computation of the program such that $A_R$ accepts the corresponding input/output sequence (which means that the specification is violated).

Our approach is not limited to programs that read input and write output in strict alternation, but extends Madhusudan’s results to the more general class of programs with bounded delay, which may read several input symbols before producing an output symbol (or vice versa).

The size of the resulting DTA is exponential in the size of the given specification automaton and doubly exponential in the number of program variables and the delay bound. We also establish a lower bound, showing that the set of all programs over $m$ Boolean variables that satisfy a given specification cannot even be recognized by an NTA with less than $2^{2^{m-1}}$ states, if any such program exists. However, note that a DTA (or NTA) accepting precisely these programs enables us to extract a minimal program for the given specification and the given set of program variables. Hence, while the tree automaton may be large, the synthesized program itself might be rather small.

3 Required Program Variables: A Lower Bound

Furthermore, we consider the question of how many (Boolean) variables are needed to satisfy a given specification. More specifically, we show that for certain specifications in LTL, at least $\Omega(2^{\sqrt{n}})$ Boolean variables are required, where $n$ is the size of the respective LTL formula. This lower bound almost matches the exponential upper bound that can be derived from the doubly exponential upper bound for the size of transition systems for a given LTL specification [6]. In order to prove this lower bound, we draw on concepts from graph theory and exploit the fact that the so-called transition graphs of structured programs over a small number of variables have small tree-width. We show that for certain specifications, the tree-width of the transition graphs of the programs satisfying these specifications must be large, which allows us to deduce a lower bound for the number of variables used by these programs.

54
Synthesizing Structured Reactive Programs via Deterministic Tree Automata

References


Satisfiability of ECTL* with constraints

Claudia Carapelle (carapelle@informatik.uni-leipzig.de)

Institut für Informatik, Universität Leipzig, Germany

Temporal logics like LTL, CTL or CTL* are nowadays standard languages for specifying system properties in model-checking. They are interpreted over node labeled graphs (Kripke structures), where the node labels (also called atomic propositions) represent abstract properties of a system. Clearly, such an abstracted system state does in general not contain all the information of the original system state. Consider for instance a program that manipulates two integer variables \( x \) and \( y \). A useful abstraction might be to introduce atomic propositions \( v_{-32},\ldots,v_{32} \) for \( v \in \{x,y\} \), where the meaning of \( v_k \) for \(-2^{32} < k < 2^{32}\) is that the variable \( v \in \{x,y\} \) currently holds the value \( k \), and \( v_{-32} \) (respectively, \( v_{32} \)) means that the current value of \( v \) is at most \(-2^{32}\) (respectively, at least \(2^{32}\)). It is evident that such an abstraction might lead to incorrect results in model-checking.

To overcome these problems, extensions of temporal logics with constraints have been studied. We first explain the idea in the context of LTL: Fix a relational structure \( \mathcal{A} \), typical examples for \( \mathcal{A} \) are number domains like the integers or rationals extended with certain relations. We add to standard LTL atomic formulas of the form \( R(X^{i_1}x_1,\ldots,X^{i_k}x_k) \), so called \textit{atomic constraints}. Here, \( R \) is one of the relations of the structure \( \mathcal{A} \), \( i_1,\ldots,i_k \geq 0 \), and \( x_1,\ldots,x_k \) are variables that range over the universe of \( \mathcal{A} \). An LTL-formula containing such constraints is interpreted over infinite paths of a standard Kripke structure, where in addition every node (state) associates with each of the variables \( x_1,\ldots,x_k \) an element of \( \mathcal{A} \) (one can think of \( \mathcal{A} \)-registers attached to the system states). A constraint \( R(X^{i_1}x_1,\ldots,X^{i_k}x_k) \) holds in a path \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \) if the tuple \( (a_1,\ldots,a_k) \), where \( a_j \) is the value of variable \( x_j \) at state \( s_i \), belongs to the \( \mathcal{A} \)-relation \( R \). In this way, the values of variables at different system states can be compared. In our example from the first paragraph, one might choose for \( \mathcal{A} \) the structure \( (\mathbb{Z},<,\equiv,=(a)_{a\in\mathbb{Z}}) \), where \(<\) is the usual order on \( \mathbb{Z} \), \( = \) is the equality relation, \( =_a \) is the unary predicate that only holds for \( a \). This structure has infinitely many predicates, which is not a problem with respect to satisfiability because any formula can only use finitely many of those predicates. Our main result actually is about an expansion of \( (\mathbb{Z},<,\equiv,=(a)_{a\in\mathbb{Z}}) \). Then, one might for instance write down a formula \( (x,\equiv_{100}(y)) \) which holds on a path if and only if there is a point of time where variable \( y \) holds the value 100 and for all previous points of time \( t \), the value of \( x \) at time \( t \) is strictly smaller than the value of \( y \) at time \( t+1 \).

In [7], Demri and Gascon studied LTL extended with constraints from a language IPC*.

If we disregard succinctness aspects, these constraints are equivalent to constraints over the structure

\[
\mathcal{Z} = (\mathbb{Z},<,\equiv,=(a)_{a\in\mathbb{Z}},(\equiv_{a,b})_{0\leq a < b}),
\]

\[1\]

*Supported by the DFG Research Training Group 1763 (QuantLA).
where \( \equiv_{a,b} \) denotes the unary relation \( \{ a + xb \mid x \in \mathbb{Z} \} \), expressing that an integer is congruent to \( a \) modulo \( b \). The main result from [7] states that satisfiability of LTL with constraints from \( \mathbb{Z} \) is decidable and in fact \( \text{PSPACE} \)-complete, and hence has the same complexity as satisfiability for LTL without constraints.

In the same way as outlined for LTL above, atomic constraints can be added as atomic path formulas to branching-time logics as CTL, CTL* and even its extension ECTL*. A weak form of CTL* with constraints from \( \mathbb{Z} \) (where only integer variables at the same state can be compared) was first introduced in [4], where it is used to describe properties of infinite transition systems, represented by relational automata. It is shown in [4] that the model checking problem for CTL* over relational automata is undecidable.

Demri and Gascon [7] asked whether satisfiability of CTL* with constraints from \( \mathbb{Z} \) over Kripke structures is decidable. This problem was investigated in [2, 9], where several partial results were shown: If we replace in \( \mathbb{Z} \) the binary predicate \( < \) by unary predicates \( <_{c} = \{ x \mid x < c \} \) for \( c \in \mathbb{Z} \), then satisfiability for CTL* has been shown decidable by [9]. For the full structure \( \mathbb{Z} \) satisfiability has been shown decidable for CEF+, the fragment of CTL* which contains the existential and universal fragment of CTL* as well as EF, see [2].

In our work we deal with ECTL* [11, 12], which is a proper extension of CTL*, where the CTL* path formulas are replaced by the set of all regular properties of paths (represented by Büchi-automata or MSO-formulas). The main result we obtain is the following:

**Theorem 1** Satisfiability of ECTL* with constraints over \( \mathbb{Z} \) is decidable.

Our proof is divided into two steps. The first step provides a tool to prove decidability of ECTL* with constraints over any structure \( A \) (called a concrete domain) over a countable signature \( \sigma \) which satisfies the property that the complement of any of its relations has to be definable in positive existential first-order logic over \( A \) (in this case we call \( A \) negation closed). Let \( L \) be a logic that satisfies the following three properties:

P.1 Satisfiability of a given \( L \)-sentence over the class of infinite node-labeled trees is decidable.

P.2 \( L \) is closed under boolean combinations with monadic second-order formulas (MSO).

P.3 \( L \) is compatible with one dimensional first-order interpretations and with the \( k \)-copy operation.

A typical such logic is MSO itself. By Rabin’s seminal tree theorem [10], satisfiability of MSO-sentences over infinite node-labeled trees is decidable, and Muchnik’s theorem (cf. [13]) implies compatibility of MSO with \( k \)-copying.

Assuming \( L \) has these properties, we prove that satisfiability of ECTL* with constraints over \( A \) is decidable if one can compute from a given finite subsignature \( \tau \subseteq \sigma \) an \( L \)-sentence \( \psi_{\tau} \) (over the signature \( \tau \)) such that for every countable \( \tau \)-structure \( B \), \( B \models \psi_{\tau} \) if and only if there is a homomorphism from \( B \) to \( A \) (i.e., a mapping from the domain of \( B \) to the domain of \( A \) that preserves all relations from \( \tau \)). We say that the structure \( A \) has the property EHD(\( L \)) if such a computable function \( \tau \mapsto \psi_{\tau} \) exists. EHD(\( L \)) stands for “existence of homomorphism is \( L \)-definable”. For instance, the structure \( (\mathbb{Q}, <, =) \) has the property EHD(MSO).

The first step of our proof can be then summarized in the following:
Satisfiability of ECTL* with constraints

**Theorem 2** Let $\mathcal{L}$ be a logic satisfying P.1-3. If the relational structure $\mathcal{A}$

- is negation closed,
- has the property $\text{EHD}(\mathcal{L})$,

then satisfiability of ECTL* with constraints over $\mathcal{A}$ is decidable.

The second step consists in showing that $\mathcal{Z}$ satisfies the conditions to apply Theorem 2. While it is easy to see that $\mathcal{Z}$ from (1) is negation closed, it is not clear whether it has the property $\text{EHD}(\text{MSO})$ (we conjecture that it does not). Hence, we need a different logic. It turns out that $\mathcal{Z}$ has the property $\text{EHD}(\text{Bool}(\text{MSO}, \text{WMSO+B}))$, where WMSO+B is the extension of weak monadic second-order logic (where only quantification over finite subsets is allowed) with the bounding quantifier $B$ and $\text{Bool}(\text{MSO}, \text{WMSO+B})$ stands for all Boolean combinations of MSO and WMSO+B sentences. A formula $BX\varphi$ holds in a structure $\mathcal{A}$ if and only if there exists a bound $b \in \mathbb{N}$ such that for every finite subset $B$ of the domain of $\mathcal{A}$ with $\mathcal{A} \models \varphi(B)$ we have $|B| \leq b$. Recently, Bojańczyk and Toruńczyk have shown that satisfiability of WMSO+B over infinite node-labeled trees is decidable [1]. Thus, WMSO+B is a candidate logic for our method. Unfortunately, WMSO+B is not closed under Boolean combinations with MSO-sentences. Thus, we consider the logic $\mathcal{L} = \text{Bool}(\text{MSO}, \text{WMSO+B})$ that consists of all Boolean combinations of MSO and WMSO+B-sentences. Fortunately, the decidability proof for WMSO+B can be extended to $\mathcal{L}$. Moreover, $\mathcal{L}$ is compatible with one-dimensional first-order interpretations and with the $k$-copy operation. Thus, we show that:

**Theorem 3** $\mathcal{L} = \text{Bool}(\text{MSO}, \text{WMSO+B})$ satisfies P.1-3, and $\mathcal{Z}$ is negation closed and enjoys the property $\text{EHD}(\mathcal{L})$.

Theorem 3 and Theorem 2 together yield Theorem 1.

While it would be extremely useful to add successor constraints ($y = x + 1$) to $\mathcal{Z}$, this would lead to undecidability even for LTL [6]. Nonetheless $\mathcal{Z}$ allows qualitative representation of increment, for example $x = y + 1$ can be abstracted by $(y > x) \land \bigvee_{i=1}^{2^k-1} (\equiv_{i,2^k}(x) \land \equiv_{i+1,2^k}(y))$ where $k$ is a large natural number. This is why temporal logics extended with constraints over $\mathcal{Z}$ seem to be a good compromise between (unexpressive) total abstraction and (undecidable) high concretion.

Since satisfiability of ECTL* (without constraints) is non-elementary (which follows from the fact that MSO over infinite words is non-elementary), the same lower complexity bound also holds for ECTL* with constraints from $\mathcal{Z}$. On the other hand, satisfiability of CTL* is 2EXPTIME-complete [8], but unfortunately, our proof does not yield any complexity bound for satisfiability of CTL* with constraints from $\mathcal{Z}$. The boolean combinations of (WMSO+B)-sentences and MSO sentences that have to be checked for satisfiability (over infinite trees) are of a simple structure, in particular their quantifier depth is not high, but no complexity statement for satisfiability of WMSO+B is made in [1], and it seems to be difficult to analyze the algorithm from [1]. It is based on a construction for cost functions over finite trees from [5], where the authors only note that their construction seems to have very high complexity.

Let us stress that the approach to decide satisfiability of ECTL* with constraints via property $\text{EHD}(\mathcal{L})$ is rather general and not restricted to be used for the integers. For
instance, this method can be applied to several “tree-like” concrete domains, like semi-linear orders, ordinal trees and trees of a fixed height. Moreover, with a slight variation the approach can also deal with finite satisfiability, i.e., with the problem whether a given formula has a model whose underlying Kripke structure is finite.

These results are part of a joint work with Alexander Kartzow and Markus Lohrey, and some of them have already appeared in [3].

References


Composition of Stochastic Timed Automata

Pierre Carlier (pierre.carlier@umons.ac.be)

LSV, CNRS & ENS Cachan, France & Université de Mons, Belgium

We consider the model of stochastic timed automata that has been introduced in [1], a model in which both delays and discrete choices are made probabilistically. We are interested in the composition of two stochastic timed automata in the case where both automata run independently. In order to define this composition, we have to find probability measures over delays and edges satisfying some properties, which are not trivial. We try to find a class of stochastic timed automata in which the composition is internal, well-defined and expresses an independent running of both automata.

1 Stochastic Timed Automata

1.1 Definition

The notion of stochastic timed automata has been introduced in [1]. We first recall the notion of timed automata. Let $X$ be a finite set of real-valued variables called clocks. A clock valuation over $X$ is a function $\nu : X \rightarrow \mathbb{R}_+$ where $\mathbb{R}_+$ is the set of non-negative real numbers. We write $\mathbb{R}_+^X$ for the set of clock valuations over $X$. Given $\nu \in \mathbb{R}_+^X$, $\tau \in \mathbb{R}_+$ and $Y \subseteq X$ we define the clock valuations $\nu + \tau$ by $(\nu + \tau)(x) = \nu(x) + \tau$ for every $x \in X$, and $[Y \leftarrow 0]\nu$ by $[Y \leftarrow 0]\nu(x) = 0$ if $x \in Y$ and $[Y \leftarrow 0]\nu(x) = \nu(x)$ otherwise. A guard over $X$ is any finite conjunction of expressions of the form $x \sim c$ where $x \in X$, $c \in \mathbb{N}$ and $\sim \in \{<, >\}$. We denote by $\mathcal{G}(X)$ the set of guards over $X$. We say that $\nu$ satisfies a guard of the form $x \sim c$ with $x \in X$, $c \in \mathbb{N}$ and $\sim \in \{<, >\}$ whenever $\nu(x) \sim c$ and we write $\nu \models g$ if $\nu$ satisfies $g$.

A timed automaton is a tuple $\mathcal{A} = (L, l_0, X, E)$ such that: (i) $L$ is a finite set of locations, (ii) $l_0 \in L$ is the initial location, (iii) $X$ is a finite set of clocks and (iv) $E \subseteq L \times \mathcal{G}(X) \times 2^X \times L$ is a finite set of edges.

The semantics of a timed automaton is a transition system $T_\mathcal{A} = (Q, q_0, \rightarrow)$ where $Q = L \times \mathbb{R}_+^X$ is the set of states, $q_0 = (l_0, 0_X)$ is the initial state, with $0_X$ being the clock valuation that assigns 0 to each clock $x$, and $\rightarrow$ is the transition relation defined as follows: given $(l, \nu)$ and $(l', \nu') \in Q$, we have $(l, \nu) \rightarrow (l', \nu')$ if there is $\tau \in \mathbb{R}_+$ and there is $e \in E$ such that $e = (l, g, Y, l')$ for some $g \in \mathcal{G}(X)$ and some $Y \subseteq X$, $\nu + \tau \models g$ and $\nu' = [Y \leftarrow 0](\nu + \tau)$. We then write $(l, \nu) \xrightarrow{\tau, e} (l', \nu')$. We define a finite run $\rho$ of $\mathcal{A}$ as a finite sequence of consecutive transitions: $\rho = q_1 \xrightarrow{\tau_1, e_1} q_2 \xrightarrow{\tau_2, e_2} \cdots \xrightarrow{\tau_n, e_n} q_{n+1}$ where, for each $i$, $q_i = (l_i, \nu_i)$ is a state. Similarly, we define an infinite run of $\mathcal{A}$ as an infinite sequence of consecutive transitions: $\rho = q_1 \xrightarrow{\tau_1, e_1} q_2 \xrightarrow{\tau_2, e_2} \cdots$. We write $\text{Runs}(\mathcal{A}, q_1)$ (resp. $\text{Runs}_f(\mathcal{A}, q_1)$) for the set of infinite (resp. finite) runs of $\mathcal{A}$ that start in $q_1$. 
One would like to extend the notion of timed automata to the notion of stochastic timed automata in order to define a probability measure over Runs($\mathcal{A}, q$) for every state $q$. In order to do it, we have to equip $\mathcal{A}$ with probability distributions over both delays and edges. We first introduce some notations. Given a state $q$ of $\mathcal{A}$ and an edge $e \in E$, we define $I(q, e) := \{ \tau \in \mathbb{R}_+ \mid \exists q' \in Q \text{ s.t. } q \xrightarrow{\tau e} q' \in \rightarrow\}$ and $I(q) := \bigcup_{e \in E} I(q, e)$. Intuitively, the set $I(q, e)$ depicts the set of delays after which, starting from $q$, edge $e$ is immediately enabled while $I(q)$ is the set of delays after which, starting from $q$, some edge $e$ is immediately enabled. In the sequel we are only interested by the reachable states and we assume that $\lambda(I(q)) > 0$ for each reachable state $q$, where $\lambda$ denotes the Lebesgue measure.

**Definition 1.1.** A stochastic timed automaton is a tuple $\mathcal{A} = (L, l_0, X, E, (\mu_q, p_q)_{q \in L \times \mathbb{R}_+^X})$ where $(L, l_0, X, E)$ is a timed automaton and:

(i) for every $q = (l, \nu) \in L \times \mathbb{R}_+^X$, $\mu_q$ is a probability distribution over $\mathbb{R}_+$ (equipped with the Borel $\sigma$-algebra) such that $\mu_q(I(q)) = 1$ and $p_q$ is a probability distribution over the set of edges enabled in $q$, i.e. over $\{(l, g, Y, l') \in E \mid \nu \models g\}$,

(ii) for every $q \in L \times \mathbb{R}_+^X$, $\mu_q$ is equivalent to the restriction of $\lambda$ on $I(q)$.

We recall that two measures $\mu$ and $\nu$ over a $\sigma$-algebra $S \subseteq 2^\Omega$ are said equivalent if for every $A \in S$, $\mu(A) = 0 \Leftrightarrow \nu(A) = 0$. We also recall that the restriction of the Lebesgue measure $\lambda$ on a Borel set $B$ of $\mathbb{R}_+$, denoted by $\lambda_B$, is defined by $\lambda_B(A) = \lambda(A \cap B)$ for every Borel set $A$.

### 1.2 A probability measure over infinite runs

Now, given a stochastic timed automaton $\mathcal{A}$, with the aim of defining a probability measure over $\text{Runs}(\mathcal{A}, q)$ for each state $q$, we introduce some notations. Given a state $q$ and a finite sequence of edges $(e_i)_{1 \leq i \leq n}$ we define the symbolic path starting from $q$ and determined by $(e_i)_{1 \leq i \leq n}$ as the set of finite runs $\pi(q, e_1, \ldots, e_n) := \{ \rho = q \xrightarrow{\tau_1, e_1} q_1 \cdots \xrightarrow{\tau_n, e_n} q_n \mid \tau_1, \ldots, \tau_n \in \mathbb{R}_+ \}$. Similarly, given a Borel set $C$ of $\mathbb{R}_+^n$, we define the constrained symbolic path starting from $q$, determined by $(e_i)_{1 \leq i \leq n}$ and satisfying $C$ as the set of finite runs $\pi_C(q, e_1, \ldots, e_n) = \{ \rho = q \xrightarrow{\tau_1, e_1} q_1 \cdots \xrightarrow{\tau_n, e_n} q_n \mid (\tau_1, \ldots, \tau_n) \in C \}$. Given a (constrained) symbolic path $\pi$, we define the cylinder generated by $\pi$, denoted by $\text{Cyl}(\pi)$, as the set of infinite runs $\rho$ such that there is a prefix of $\rho$ that is in $\pi$. In other words, if $\pi = \pi_C(q, e_1, \ldots, e_n)$ where $q \in Q$, $e_1, \ldots, e_n \in E$ and $C$ is a Borel set of $\mathbb{R}_+^n$, then $\text{Cyl}(\pi) = \{ \rho \in \text{Runs}(\mathcal{A}, q) \mid \rho = q \xrightarrow{\tau_1, e_1} q_1 \cdots \xrightarrow{\tau_n, e_n} q_n \rightarrow \cdots \text{ and } (\tau_1, \ldots, \tau_n) \in C \}$.

We inductively define a measure, denoted by $P_{\mathcal{A}}$, over finite symbolic paths from state $q$ by:

$$P_{\mathcal{A}}(\pi(q, e_1, \ldots, e_n)) = \int_{t \in I(q, e_1)} p_{q+t}(e_1)P_{\mathcal{A}}(\pi(q_t, e_2, \ldots, e_n))d\mu_q(t)$$

where $e_1, \ldots, e_n$ are in $E$ and $q_t$ is such that $q \xrightarrow{t} q + t \xrightarrow{e_1} q_t$, and we initialize with $P_{\mathcal{A}}(\pi(q)) = 1$. The formula for $P_{\mathcal{A}}$ relies on the fact that the probability of taking transition $e_1$ at time $t$ coincides with the probability of waiting $t$ time units and then choosing $e_1$ among the enabled transitions, i.e. $p_{q+t}(e_1)d\mu_q(t)$. The value of $P_{\mathcal{A}}(\pi)$, where $\pi = \pi(q, e_1, \ldots, e_n)$, is the result of $n$ successive integrals. Hence one can extend the...
measure $\mathbb{P}_A$ to the constrained symbolic paths. This extension is required to measure rather complex sets. Now, since we are interested by the set of infinite runs, one can extend $\mathbb{P}_A$ to the cylinders by $\mathbb{P}_A(Cyl(\pi)) = \mathbb{P}_A(\pi)$, where $\pi$ is a (constrained) symbolic path. Using some extension’s theorem as the Carathéodory’s theorem, we can extend $\mathbb{P}_A$ in a unique way to the $\sigma$-algebra generated by the cylinder starting in $q$, which we note $\Omega^q_A$.

Proposition 1.2 ([2]). Let $A = (L, l_0, X, E, (\mu_q, p_q)_{q \in L \times \mathbb{R}^X})$ be a stochastic timed automaton. For every state $q \in Q$, $\mathbb{P}_A$ is a probability measure over $(\text{Runs}(A, q), \Omega^q_A)$.

2 Composition

2.1 Construction and definition

We want now to define the composition of two stochastic timed automata. We first define the composition of two timed automata. If $A_1 = (L_1, l_0^{(1)}, X_1, E_1)$ and $A_2 = (L_2, l_0^{(2)}, X_2, E_2)$ are two timed automata such that $X_1 \cap X_2 = \emptyset$, we define the timed automaton $A_1 \times A_2$ as the tuple $A_1 \times A_2 = (L_1 \times L_2, (l_0^{(1)}, l_0^{(2)}), X_1 \cup X_2, E)$, where $E = E_{1,1} \cup E_{1,2}, E_{1,1} = \{(l_1, l_2), g, Y, (l'_1, l_2) \} \mid (l_1, g, Y, l'_1) \in E_1, l_2 \in L_2\}$, and $E_{1,2}$ is defined similarly. We abusively denote $E_{1,1}$ by $E_{1,1}$ and $E_{1,2}$ by $E_{1,2}$.

Now one would like to extend the composition to the stochastic timed automata. Let $A_1 = (L_1, l_0^{(1)}, X_1, E_1, (\mu^{(1)}, p^{(1)}), \nu^{(1)}) \subset L_1 \times \mathbb{R}^X_{+}$ and $A_2 = (L_2, l_0^{(2)}, X_2, E_2, (\mu^{(2)}, p^{(2)}) \subset L_2 \times \mathbb{R}^X_{+}$ be two stochastic timed automata. We want to find a family of couples of probability measures, $(\mu_q, p_q)_{q \in L_1 \times L_2 \times \mathbb{R}^X_{+} \times \mathbb{R}^X_{+}}$, in order to define the stochastic timed automaton $A_1 \times A_2 = (L, l_0, X, E, (\mu_q, p_q)_{q \in L \times \mathbb{R}^X})$, where $(L, l_0, X, E)$ is as defined in the previous definition and thus, $L \times \mathbb{R}^X = L_1 \times L_2 \times \mathbb{R}^X_{+} \times \mathbb{R}^X_{+}$. For any state $q = ((l_1, l_2), (\nu_1, \nu_2)) \in L_1 \times L_2 \times \mathbb{R}^X_{+} \times \mathbb{R}^X_{+}$ we have to choose a probability distribution $\mu_q$ over $I(q)$ and a probability distribution $p_q$ over the set of enabled edges in $q$, i.e. the set $\{(l_1, l_2), g, Y, (l'_1, l_2) \} \subset E_{1,1}$ (resp. $\nu_1)$ and $\nu_2$) is a state of $A_1 \times A_2$ we write $q_1$ (resp. $q_2$) for the projection of $q$ in $A_1$ (resp. $A_2$), i.e. $q_1 = (l_1, \nu_1)$ (resp. $q_2 = (l_2, \nu_2)$). We write $f_{q,1}$ (resp. $f_{q,2}$) for the Radon-Nikodym derivative of $\mu_q$ (resp. $\mu_q^{(2)}$) with respect to the Lebesgue measure $\lambda$, i.e. $\mu_q^{(1)}(A) = f_{q,i}(t)dt$ for each Borel set $A$ of $\mathbb{R}^X$ and each $i \in \{1, 2\}$. We have that $f_{q,i}$ is the density function of $\mu_q$ and we define $F_{q,i}$ for the cumulative function associated to $f_{q,i}$ for each $i \in \{1, 2\}$. We then write $X_{q,1}$ (resp. $X_{q,2}$) for a random variable of density $f_{q,1}$ (resp. $f_{q,2}$) and where $X_{q,1}$ and $X_{q,2}$ are independent.

We define $\mu_q(A) = \int_A f_{q,\min}(t)dt$ for each Borel set $A$, where $f_{q,\min}$ is the density function of $\min(X_{q,1}, X_{q,2})$. One can show that $f_{q,\min}(t) = f_{q,1}(t)(1 - F_{q,2}(t)) + f_{q,2}(t)(1 - F_{q,1}(t))$ for every $t \geq 0$. In order to define the probability distributions $p_q$ over the enabled edges in $q$, one could consider that from state $q$, both systems $A_1$ and $A_2$ are in a race to win the next edge, i.e. $A_1$ wins the race if the first edge taken from $q$ is in $E_1$. Hence, given $t \in I(q)$, and an edge $e \in E_1$ enabled in $q + t$, one would like that $p_{q,t+}(e) = w_q^1(t)\nu_{1,q}(e)$ where $w_q^1(t)$ is the probability that, starting from $q$, $A_1$ wins the race knowing that it was won after a delay of $t$ time units. Formally, we define $w_q^1(t) = \lim_{\varepsilon \to 0} \mathbb{P}(X_{q,1} = \min(X_{q,1}, X_{q,2}) \mid \min(X_{q,1}, X_{q,2}) \in [t, t + \varepsilon])$ for every $t \geq 0$ and
we then can show that \( w^1_q(t) = \frac{f_{q,1}(t)(1-F_{q,2}(t))}{f_{q,\min}(t)} \) if \( f_{q,\min}(t) \neq 0 \). We formalize this in the next definition.

Let CSTA denote the class of stochastic timed automata \( \mathcal{A} \) such that for every state \( q \),

(a) the density function \( f_q \) associated with \( \mu_q \) is almost-surely continuous on \( \mathbb{R}_+ \), and

(b) for every \( t, t' \geq 0 \) with \( t \) and \( t+t' \) in \( I(q) \), \( f_q(t+t')(1-F_q(t+t')) = f_{q+t}(t'')(1-F_q(t+t'')) \), where \( F_q \) (resp. \( F_{q+t} \)) is the cumulative function associated with \( f_q \) (resp. \( f_{q+t} \)).

**Definition 2.1.** Let \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) be two stochastic timed automata as before. We say that \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) are *composable* if \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) are in CSTA and if \( X_1 \cap X_2 = \emptyset \) and we define the composition \( \mathcal{A}_1 \times \mathcal{A}_2 \) as the stochastic timed automaton \( \mathcal{A}_1 \times \mathcal{A}_2 = (L, l_0, X, E, (\mu_q, p_q)_{q \in L \times \mathbb{R}_+} \) where, for any \( q = ((l_1, l_2), (\nu_1, \nu_2)) \),

(i) \((L, l_0, X, E)\) is the composition of the underlying timed automata \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \),

(ii) \( \mu_q \) is defined as follows: \( \forall A \in B(\mathbb{R}_+) \), \( \mu_q(A) = \int_A f_{q,\min}(t)dt \), where \( f_{q,\min}(t) = f_{q,1}(t)(1-F_{q,2}(t)) + f_{q,2}(t)(1-F_{q,1}(t)) \) for every \( t \geq 0 \), and

(iii) for any \( t \in I(q) \), \( p_{q+t} \) is defined as follows: \( p_{q+t}(e) = \mathbb{1}_{E_1}(e)w^1_q(t)p_{q,t}(1) + \mathbb{1}_{E_2}(e)w^2_q(t)p_{q,t}(2) \) for every \( e \in E \), where for any \( t \in I(q) \),

\[
w^1_q(t) := \frac{f_{q,1}(t)(1-F_{q,2}(t))}{f_{q,\min}(t)} \quad \text{and} \quad w^2_q(t) := \frac{f_{q,2}(t)(1-F_{q,1}(t))}{f_{q,\min}(t)}
\]

if \( f_{q,\min}(t) \neq 0 \), and \( w^1_q(t) = w^2_q(t) = 0 \) if \( f_{q,\min}(t) = 0 \).

Hypotheses (a) and (b) are needed so that for every state \( q \), \( f_{q,\min} \), \( w^1_q \) and \( w^2_q \) are well-defined. One can notice that the hypotheses express conditions that do not depend of the distributions of both automata simultaneously, it is independent of the automaton product. Hypothesis (b) comes from the fact that, given a state \( q, p_q \) is well-defined if for any \( t, t' \geq 0 \), \( p_{q+t+t'} = p_{q+(t+t')} \) since \( (q+t) + t' = q + (t+t') \). One can show that this equality holds if \( w^i_q(t+t') = w^i_q(t') \) for any \( t, t' \geq 0 \) and for any \( i \in \{1, 2\} \). We can then show that it suffices that hypothesis (b) holds to have this last equality. One can show that if \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \in \text{CSTA} \) then \( \mathcal{A}_1 \times \mathcal{A}_2 \in \text{CSTA} \).

It is not difficult to show that uniform and exponential distributions satisfy hypothesis (a) and (b). One can thus see that two stochastic timed automata \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \), equipped with uniform or exponential distributions over the delays, are composable. We now give an example of probability measures that do not satisfy hypothesis (b).

**Example 2.2 (Weighted uniform distributions).** If for any \( q = (l_1, \nu_1) \) with \( \nu_1 \in [0, 1] \), we have that for any \( t \geq 0 \),

\[
f_{q,1}(t) = \frac{3}{2} - \frac{1}{\nu_1} \mathbb{1}_{[0,1/\nu_1]}(t) + \frac{1}{2} - \frac{1}{\nu_1} \mathbb{1}_{[1/\nu_1, 1]}(t).
\]

Then one can easily show that \( F_{q,1}(t) = \frac{3}{2} - \frac{t}{\nu_1} \mathbb{1}_{[0,1/\nu_1]}(t) + \frac{1}{2} + \frac{t}{\nu_1} \mathbb{1}_{[1/\nu_1, 1]}(t) + \mathbb{1}_{[1-\nu_1, +\infty]}(t) \). Now, let us assume that \( q = (l_1, 0) \), \( t = \frac{1}{2} \) and \( t' = \frac{1}{5} \). Then \( t \) and \( t+t' \) are in \( I(q) \) and we can show that \( f_{q,1}(5/8)(1-F_{q+(1/2),1}(1/8)) = 5/16 \) and \( f_{q+(1/2),1}(1/8)(1-F_{q,1}(5/8)) = 9/16 \). We conclude that hypothesis (b) is not satisfied for some weighted uniform distribution and thus, a stochastic timed automaton equipped with this weighted uniform distribution over the delays, can not be composed with another stochastic timed automaton.
2.2 Independence

Now, in order to express an independent running of $A_1$ and $A_2$, one would like for example that, given a state $q = ((l_1, l_2), (\nu_1, \nu_2))$, the probability that $e_1$ is the first edge in $E_1$ taken in $A_1 \times A_2$ from $q$ coincides with the probability that $e_1$ is the first edge taken in $A_1$ from $(l_1, \nu_1)$:

$$P_{A_1 \times A_2}(q \xrightarrow{A_2^*} e_1 \xrightarrow{\cdot}) = P_{A_1}(q \xrightarrow{} e_1 \xrightarrow{\cdot}).$$  (1)

where $q \xrightarrow{A_2^*} e_1 = \bigcup_{n \in \mathbb{N}} \bigcup_{(f_1, \ldots, f_n) \in E_2^n} Cyl(\pi(q, f_1, \ldots, f_n, e_1))$. One can show that (1) is satisfied if $A_1$ and $A_2$ are almost-surely non-zeno and if for each $i \in \{1, 2\}$, for each state $q$ of $A_1 \times A_2$, for each $t, t' \geq 0$,

$$f_{q,i}(t + t') = (1 - F_{q,i}(t))f_{q+t,i}(t').$$  (2)

We recall that an infinite run $\rho = q_1 \xrightarrow{t_1, e_1} q_2 \xrightarrow{t_2, e_2} q_3 \ldots$ is zeno if $\sum_{i \geq 1} t_i < +\infty$ and that the set of zeno runs can be expressed by means of cylinders. Moreover, one can show that if $C$ is the class of stochastic timed automata almost-surely non-zeno, satisfying hypothesis (a) and (b) of Definition 2.1 and satisfying (2), then the composition is internal and well-defined in $C$, and satisfies (1).

3 Future Work

Since we are interested by the model-checking problem, we would like now to find an interesting class of stochastic timed automata in which, given an automaton $A$ and an LTL-formula $\varphi$, we can decide whether the probability that $A$ satisfies $\varphi$ is 1 or not. We try to find such an interesting class, satisfying all hypothesis of the class $C$, such that the composition is internal and well-defined in $C$, and satisfies (1). Several classes of stochastic timed automata in which we can decide the almost-sure model-checking problem are described in [2] and [3].

We are also currently working on a notion of bisimulation in stochastic timed automata. The objective is to have a bisimulation that is a congruence with regard to the composition.

References


1 Introduction

The Satisfiability-Modulo-Theories (SMT) problem addresses checking the satisfiability of SMT formulas, i.e., Boolean combinations of constraints of one or more theories. SMT solvers use a SAT solver to find satisfying solutions for the Boolean skeleton of an input SMT formula, which are in turn checked for consistency for the underlying theories with other decision procedures.

The last decade has brought great achievements in the field of SMT solving. For instance, the SMT-LIB standard defines a common input format for SMT solvers and provides the community with benchmarks for different theories. In addition, SMT competitions motivate the development and improvement of SMT solvers. Nowadays, different efficient SMT solvers are available for several theories, e.g., for linear real arithmetic. However, only a few solvers support nonlinear real arithmetic (NRA) and nonlinear integer arithmetic (NIA), the theory of the reals and integers with addition and multiplication.

Nonlinear real arithmetic was shown to be decidable by Tarski [17]. Though the worst-case time complexity of solving real arithmetic formulas is doubly exponential in the number of variables [19], its existential fragment, which is addressed by SMT solving, can be solved in exponential time [12]. One of the most widely used decision procedures for NRA is the cylindrical algebraic decomposition (CAD) method [7]. Other well-known methods use, e.g., Gröbner bases (GB) [18] or the realization of sign conditions [2]. Some incomplete methods based on, e.g., interval constraint propagation (ICP) [11] and virtual substitution (VS) [20], can handle significant fragments and, even though they have the same worst-case complexity as the complete methods, they are more efficient in practice. Moreover, they are well suited to be complemented by complete methods, to which they pass reduced sub-problems.

Nonlinear integer arithmetic is not decidable, however there are some approaches which try to tackle this problem. Procedures which are tailored for NRA can always be used to find the unsatisfiability of the real relaxation of the given NIA instance. Furthermore, ICP can be adapted to be used for NIA and we can transform an NIA formula into a formula in propositional logic, if we consider only the decidable fragment where the domains of all variables are bounded (bit-blasting [6]).

Most activities in the area of SMT solving focus on theories such as equality logic, bitvectors, arrays, uninterpreted functions or linear arithmetic over the reals and integers. This research resulted in SMT solvers like, e.g., CVC3 [1], MathSAT [5] or OpenSMT [4] just
to name a few. However, less activity can be observed for SMT solvers for NRA, as the underlying procedures are very complex: besides some incomplete solvers like MiniSmt [21] and iSAT3 [11, 16], we are aware of only one SMT solver, i.e., Z3 [13], that is complete for NRA. Even less activity is observed in NIA: to the best of our knowledge, only Z3 and the SMT solving spin-off of Aprove [6] implementing bit-blasting can tackle this logic.

The development of a complete SMT solver for NRA is problematic because the aforementioned NRA and NIA decision procedures are not SMT-compliant, i.e., they do not fulfill the requirements for an embedding into an efficient SMT solver. Firstly, in less-lazy SMT solving, theory solvers should be able to work incrementally, i.e., if they determine the satisfiability of a set of constraints, they should be able to check an extended set of constraints on the basis of the previous result. Secondly, in case a constraint set is unsatisfiable, theory solvers should be able to compute an infeasible subset as explanation. Thirdly, they must be able to backtrack according to the search of the SAT solver.

2 Satisfiability modulo real and integer arithmetic

SMT solving denotes an algorithmic framework for solving Boolean combinations of constraints from some theories. SMT solvers combine a SAT solver computing satisfying assignments for the Boolean structure of the SMT formula with procedures to check the consistency of theory constraints. For more details on SMT we refer to [3, Ch. 26].

We consider NRA/NIA formulas $\varphi$, which are Boolean combinations of constraints $c$ comparing polynomials $p$ to 0. A polynomial $p$ can be a constant, a variable $x$, or a composition of polynomials by addition, subtraction or multiplication:

\[
\begin{align*}
p & ::= 0 \mid 1 \mid x \mid (p + p) \mid (p - p) \mid (p \cdot p) \\
c & ::= p = 0 \mid p < 0 \mid p > 0 \\
\varphi & ::= c \mid (\neg \varphi) \mid (\varphi \land \varphi) \mid (\exists x \varphi)
\end{align*}
\]

The semantics of NRA/NIA formulas is defined as usual.

Given a polynomial $p = a_1 x_1^{e_{1,1}} \cdots x_n^{e_{1,n}} + \cdots + a_k x_1^{e_{k,1}} \cdots x_n^{e_{k,n}}$ in monomial normal form, by\(\deg(p) := \max_{1 \leq j \leq k} (\sum_{i=1}^n e_{i,j})\) we denote the degree of $p$. We call an NRA/NIA formula $\varphi$ linear if $\deg(p) \leq 1$ for all polynomials $p$ in $\varphi$, and nonlinear otherwise.

3 The design of SMT-RAT

SMT-RAT is a C++ library consisting of a collection of SMT-compliant implementations of methods for solving NRA/NIA formulas we refer to as modules. These modules can be combined to (1) a theory solver in order to extend the supported logics of an existing SMT solver by NRA/NIA (see Figure 2) or (2) an SMT solver for NRA/NIA (see Figure 1). The latter is especially intended to be a testing environment for the development of SMT-compliant implementations of further methods tackling NRA/NIA. Here, the developer only needs to implement the given interfaces of an SMT-RAT module and does not need to care about parsing input files, transforming formulas to conjunctive normal form or embedding a SAT solver in order to solve the Boolean skeleton of the given formula. Instead, SMT-RAT provides this and more features, e.g., lemma exchange.
SMT-RAT defines three types of components: the \textit{manager}, the \textit{strategy} and, as already mentioned, \textit{modules}. In addition, a front-end (1) provides the interfaces to an external SMT solver or (2) parses an input file storing an NRA/NIA formula, which we denote in the following simply as formula.

A module $m$ contains a set of formulas, called its \textit{set of received formulas} and denoted by $C_{rcv}(m)$. The main procedure of a module is \texttt{check()} and either decides whether $C_{rcv}(m)$ is satisfiable or not returning \texttt{sat} or \texttt{unsat}, respectively, or returns \texttt{unknown}. Note, that a set of formulas is semantically defined by their conjunction. We can manipulate the set of received formulas by adding (removing) formulas $\varphi$ to (from) it with \texttt{add}(\varphi) (\texttt{remove}(\varphi)). Usually, $C_{rcv}(m)$ is only changed slightly between two consecutive \texttt{check()} calls, hence, the solver’s performance can be significantly improved if the modules work incrementally and support backtracking. In case the module determines the unsatisfiability of $C_{rcv}(m)$, it is expected to compute at least one preferably small \textit{infeasible subset} $C_{inf}(m) \subseteq C_{rcv}(m)$. Moreover, a module has the possibility of naming lemmas, which are formulas being tautologies, that is they hold for all assignments of the variables occurring in them. These lemmas should encapsulate information which can be extracted from a module’s internal state and propagated among other SMT-RAT modules. Furthermore, \texttt{SMT-RAT} provides the feature that a module itself can ask other modules for the satisfiability of a set of formulas, called its \textit{set of passed formulas} denoted by $C_{pas}(m)$, using the procedure \texttt{runBackends()} which is controlled by the manager.

\texttt{SMT-RAT} allows a user to decide how to compose the different procedures, which are implemented in the modules. For this purpose we provide a graphical user interface, where the user can create such a composition we call strategy. A \texttt{strategy} is a directed tree $T := (V, E)$ with a set $V$ of (instances of) modules as nodes and $E \subseteq V \times \Omega \times V$, where $\Omega$ is a set of conditions. The \textit{manager} contains the strategy and the input formula.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig1.png}
\caption{A snapshot of an SMT-RAT composition being an SMT solver.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig2.png}
\caption{A snapshot of an SMT-RAT composition being a theory solver embedded in an SMT solver.}
\end{figure}
$C_{\text{input}}$, either received by a prefixed solver or parsed from an example file. Furthermore, it maintains the allocation of modules as follows.

Initially, the manager calls the method check() of the module $m_r$ given by the root of the strategy with $C_{\text{rcv}}(m_r) = C_{\text{input}}$ being the set of received formulas of this module. Whenever a module $m \in V$ calls runBackends(), with $C_{\text{pas}}(m)$ being its set of passed formulas, the manager calls check() of each module $m'$ with $C_{\text{rcv}}(m') = C_{\text{pas}}(m)$ being its set of received formulas, for which an edge $(m, \omega, m') \in E$ exists such that $\omega$ holds for $C_{\text{pas}}(m)$, and passes the results back to $m$. Furthermore, it also passes back the infeasible subsets and lemmas provided by the invoked modules. The module $m$ can now benefit in its solving and reasoning process from this shared information.

A condition $\omega \in \Omega$ on formulas is an arbitrary Boolean combination of formula properties, such as propositions about the Boolean structure of the formula, e.g., whether it is in conjunctive normal form (CNF), about the constraints, e.g., whether it contains equations or about the polynomials, e.g., whether they are linear.

There are already various procedures for NRA and NIA implemented in SMT-RAT, all being SMT-compliant. One module, for instance, applies an efficient CNF transformation. Another module abstracts its input NRA/NIA formula in CNF to propositional logic and applies the efficient SAT solver minisat [10]. One module uses the Simplex method combined with branch-and-bound and cutting-plane procedures as presented in [9]. We apply it on the linear constraints of any conjunction of NRA and NIA constraints. For a conjunction of NRA constraints and the real relaxation of NIA constraints SMT-RAT provides modules implementing the GB [14], VS [8] and CAD [15] procedures.

References


Model checking has become an increasingly popular technique to determine whether a system’s behavior conforms to its specification. One of the main features responsible for its success is the ability to generate counterexamples if the desired property is not guaranteed by the model, since they constitute an explanation as to why the verification task failed [4]. Based on such a counterexample, the system designer can then track down the erroneous behavior and refine the system in order to meet the specification. Furthermore, counterexample-guided abstraction refinement, one of the most successful automated abstraction refinement schemes, relies on the analysis of counterexamples to refine an over-approximation of the model [3, 8, 2].

When checking a system’s conformity to the specification, one often wants to ensure that the system may never reach a set of “bad states” $T$. For “regular”, qualitative, models, a counterexample for such a safety property is given by a finite execution fragment that takes the system from an initial state to one of the states in $T$. However, for systems whose behavior is inherently probabilistic, like randomized algorithms or realistic hardware models, it is often impossible to guarantee that a bad state is never reached. However, this may be tolerable as long as the probability for that “bad” event to occur is below a given threshold. For example, a system may behave correctly at least $99\%$ of the time, but not strictly always. Therefore, a reachability property in the probabilistic setting consists of a set of bad states $T$ together with a probability bound $\lambda$ that must not be exceeded.

In this setting, a finite trace does not necessarily constitute a violation, since its probability mass may be smaller than $\lambda$. Sets of finite paths may now be used to form a counterexample, but it can be shown that the sizes of these sets may be doubly exponentially large in the number of states of the system [7]. To alleviate this problem, the notion of critical subsystems was proposed [12]: by selecting a (hopefully small) number of states of the system, implicitly all paths only visiting the selected states are selected. This way, infinitely many “bad” paths that together exceed the probability bound $\lambda$ can be represented using finitely many states. In practice, critical subsystems are often large and can therefore be of limited use in the debugging process [9].

Instead of explicitly listing all transitions at the state-space level, system models are typically specified in a high-level formalism such as, e.g., process algebras. For probabilistic models, one typically employs the modelling language of the well-known probabilistic model checker PRISM [10], which is based on Alur and Henzinger’s reactive modules [1], to formally capture the behavior of the system. In this language, an input model consists of several modules that define variables and commands. The valuations of the variables then
form the state space of the model, while the commands induce (probabilistic) transitions between the states. Clearly, a representation of a counterexample in terms of fragments of the (high-level) system model promises to be succinct as well as easily accessible by the system designer.

We therefore aim at computing the smallest set of commands of a PRISM model that induces a critical subsystem. Such a set is called a minimal critical command set [13]. The problem of finding a minimal critical command set is NP-hard in the number of states of the model and can be solved via a formulation as a mixed-integer linear program (MILP) [13]. However, even when applying optimizations and using state-of-the-art commercial solvers, only models of moderate size can be handled and for most models the technique cannot solve the problem within reasonable time [13]. We propose a new technique to compute minimal critical command sets that scales to systems with millions of states.

Our new approach is based on enumerating sets of commands in ascending size. Starting with the empty set, larger and larger sets of commands (“hypotheses”) are enumerated. Given a hypothesis, we can restrict the high-level description of the model to the chosen set of commands and thereby obtain a sub-model $M_{|C}$ of the original model $M$. A probabilistic model checker can then be used to compute the probability to reach a state in $T$ in $M_{|C}$. If the probability mass exceeds $\lambda$, the current hypothesis is an optimal solution, since all smaller sets have been enumerated earlier. If, on the other hand, the reachability probability is below $\lambda$, the enumeration process is continued with the next hypothesis.

Clearly, in order to make this enumeration process efficient in practice, as many suboptimal solutions as possible must be pruned before they are enumerated. To achieve this, we derive constraints from the high-level representation of the model that are known to be true in any optimal solution but potentially rule out a lot of suboptimal solutions. Given such a set of constraints, a MAXSAT solver [6] can be used to find the smallest set of commands that respects these constraints. A schematic of this approach is depicted in Figure 1. Starting with a set of constraints that can be statically derived from the high-
level input model $\mathcal{M}$, the MaxSAT solver is used to find the smallest set of commands $C$ that is a potential solution to the minimal critical command set problem. If a model checker confirms that the hypothesis $C$ exceeds the probability threshold $\lambda$, it is in fact the smallest critical command set. If, however, $C$ does not suffice to violate the reachability property, the hypothesis is analyzed and additional constraints are derived that (i) rule out the current hypothesis and (ii) try to eliminate as many suboptimal solutions from the hypothesis enumeration process as possible.

We implemented the new approach in our model checking framework and performed experiments using a selection of four well-known benchmark models from Prism’s website [11] to evaluate its effectiveness. The data shows that the MaxSAT-based approach outperforms the MILP-based approach on all instances. More concretely, it achieves a speed-up of up to five orders of magnitude, consistently uses one order of magnitude less memory and therefore is able to scale to systems with millions of states [5].

We believe that our technique together with the conciseness of high-level counterexamples complements existing counterexample representations in the probabilistic setting. Unlike the MILP-based approach, it can be applied to the broader range of monotonic properties by introducing appropriate constraints. Additionally, the runtime can be further improved by exploiting parallel computing and developing more sophisticated analysis techniques for insufficient hypotheses that allow for pruning even more suboptimal solutions.

References


1 Introduction

Random distributions of practical significance can often be expressed as probabilistic functional programs. When studying a random distribution, it is often desirable to determine its probability density function (PDF). This can be used to e.g. determine the expectation or sample the distribution with a sampling method such as Markov-chain Monte Carlo (MCMC).

Bhat et al. [2] presented a compiler that computes the probability density function of a program in the probabilistic functional language Fun. Fun is a small functional language with basic arithmetic, Boolean logic, product and sum types, conditionals, and a number of built-in discrete and continuous distributions. It does not support lists or recursion. They evaluated the compiler on a number of practical problems and concluded that it reduces the amount of time and effort required to model them in an MCMC system significantly compared to hand-written models. A correctness proof for the compiler is sketched.

Bhat et al. [1] stated that their eventual goal is the formal verification of this compiler in a theorem prover. We have verified such a compiler for a similar probabilistic functional language in the interactive theorem prover Isabelle/HOL [7, 8]. Our contributions are the following:

- a formalisation of the source language, target language (whose semantics had previously not been given precisely), and the compiler on top of a foundational theory of measure spaces
- a formal verification of the correctness of the compiler
- executable code for the compiler using Isabelle’s code generator

In the process, we uncovered an incorrect generalisation of one of the compiler rules in the draft of an extended version of the paper by Bhat et al. [3].

The complete formalisation is available online [5]. For a detailed account of the formalisation, see the 2015 ESOP paper [6] or the Master’s thesis that is the basis of this work [4].
2 Source and Target Language

The source language used in the formalisation was modelled after the language Fun described by Bhat et al. [2]; similarly, the target language is almost identical to the target language used by Bhat et al. However, we have made the following changes in our languages:

- Variables are represented by de Bruijn indices.
- No sum types are supported. Consequently, the match-with construct is replaced with an IF-THEN-ELSE. Furthermore, booleans are a primitive type rather than represented as unit + unit.
- The type double is called real and it represents a real number with absolute precision as opposed to an IEEE 754 floating point number.

2.1 Types, values, and operators

The source language and the target language share the same type system and the same operators. Figure 1 shows the types and values that exist in our languages. Additionally, standard arithmetical and logical operators exist.

2.2 Source language

```
datatype pdf_type =
  UNIT | B | Z | R | pdf_type × pdf_type

datatype val =
  UnitVal | BoolVal bool | IntVal int | RealVal real | <|val, val|>

datatype pdf_operator =
  Fst | Snd | Add | Mult | Minus | Less | Equals | And | Or | Not | Pow |
  Fact | Sqrt | Exp | Ln | Inverse | Pi | Cast pdf_type
```

Figure 1: Types and values in source and target language

Figure 2 shows the syntax of the source language. It contains variables (with de Bruijn indices), values, LET-expressions (again de Bruijn), operator application, pairs, sampling
a parametrised built-in random distribution, IF-THEN-ELSE and failure. We support the same distributions as Bhat et al. (Bernoulli, uniform, Poisson, Gaussian), except for the Beta and Gamma distributions (merely because we have not formalised them yet).

The informal semantics of the source language should be largely obvious. Every source language expression returns a sub-probability distribution; the Random construct returns a random value taken from one of the built-in distributions, parametrised with another expression (e.g. mean/standard deviation for Gaussian distributions). The Fail construct returns the null measure, effectively returning no result at all. For the formal semantics, refer to one of the previously-mentioned sources. [4, 5, 6]

2.3 Target language

The target language is again modelled very closely after the one by Bhat et al. [2]. The type system and the operators are the same as in the source language. The key difference is that the Random construct has been replaced by an integral. As a result, while expressions in the source language return a measure space, expressions in the target language always return a single value. Figure 3 shows the syntax of the target language.

3 Definition and Correctness Proof of the Compiler

The correctness proof is done in two steps using a refinement approach: first, we define and prove correct an abstract compiler that returns the density function as an abstract mathematical function. We then define an analogous concrete compiler that returns a target-language expression and show that it is a refinement of the abstract compiler, which will allow us to lift the correctness result from the latter to the former.

As a first step, we implemented an abstract density compiler as an inductive predicate. We proved soundness for the abstract compiler w.r.t. the semantics of the source language, i.e. given a well-typed expression in the source language, the abstract compiler returns a density function of the distribution of the expression’s result.

The concrete compiler is another inductive predicate, modelled directly after the abstract compiler, but returning a target-language expression instead of a HOL function. We use a standard refinement approach to relate the concrete compiler to the abstract one. We thus lift the soundness result on the abstract compiler to an analogous result on the concrete compiler.
This shows that the concrete compiler always returns a well-formed target-language expression that represents a density for the sub-probability space described by the source language.

## 4 Conclusion

We formalised the semantics of a probabilistic functional programming language with predefined probability distributions and a compiler that returns the probability distribution that a program in this language describes, based on the work by Bhat et al. Then we formally verified the correctness of this compiler w. r. t. the semantics of the source and target languages.

This shows not only that the compiler given by Bhat et al. is correct (apart from the incorrect generalisation we mentioned earlier), but also that a formal correctness proof for such a compiler can be done with reasonable effort and that Isabelle/HOL in general and its measure theory library in particular are suitable for it. A useful side effect of our work was the formalisation of the Giry Monad, which is useful for formalisations of probabilistic computations in general.

**Acknowledgements.** Johannes Hötlz and Tobias Nipkow were coauthors of the 2015 ESOP paper [6] and were the author’s advisor (resp. supervisor) for the Master’s thesis on which this work is based [4].
References


Relaxing Description Logics Queries using Similarity Measures

Andreas Ecke (ecke@tcs.inf.tu-dresden.de)

DFG Research Training Group 1763
“Quantitative Logics and Automata”
Technische Universität Dresden

This is joint work with Anni-Yasmin Turhan and Rafael Peñaloza.

Description Logic (DL) knowledge bases are formal vocabularies that describe categories or specific subjects from application domains—such as for service matching, the bio-medical or geo-spatial field. The concepts in the knowledge base are characterized by relationships to other concepts using constructors available in the DL in which the knowledge base is formulated. Traditionally, DL reasoning systems only support crisp inference services, like subsumption and instances queries. The latter can be effectively used to perform different types of search tasks: Given an ABox, which describes individual objects in terms of the defined concepts, an instance query returns all those individuals that are instance of the query concept \( Q \), rejecting all others.

However, often it is also interesting to consider those individuals that are not instances: Are they completely different to \( Q \) or how similar are they to \( Q \)? In cases where the original query does not retrieve any resulting individuals, those individuals that are ‘very close’ to being an instance can still be a good alternative. The instance queries that do not only return the instances but also those that nearly match the query concept are called relaxed instance queries [2]. A natural way to relax instance queries is by using concept similarity measures (CSMs). Such a measure \( \sim \) is a function that assigns to each pair of concepts a similarity value between 0 and 1. Together with a fixed threshold \( t \), the instance query can be relaxed by returning all individuals that are instance of a concept with a similarity value of at least \( t \) to the query concept w.r.t. \( \sim \). One advantage of using CSMs as a parameter for this inference is that they can implement different notions of similarity, and regard certain features more important than others. This allows to relax queries with respect to certain features, but leave others fixed (see Figure 1).

Example 1. Humans can estimate the similarity between different animals, and actually use this similarity to reason about animals. For example, if one sees a new animal and wants to guess whether this animal can fly, one would check if similar animals are able to fly or not. But of course, for this application, certain features are more important than other: One would not compare animals with regards to their color, but rather compare if they both have wings (and how large these wings are), and if the animal is light enough. Those features are much more important to assess whether an animal can fly. The resulting similarity measure could be called \( \sim_{flying} \).
Of course, for different applications, we would use other similarity measures. To guess the diet of a new animal, we would rather compare the form of the jaws and teeth of animals and maybe compare their claws. This similarity measure might be called \( \sim_{\text{diet}} \).

Both these measures can be used to relax queries. \( \sim_{\text{flying}} \) will then yield results that are similar to the query w.r.t. the ability to fly, while allowing to relaxing other features; \( \sim_{\text{diet}} \) will yield results similar to the query w.r.t. the diet.

In this work, we consider the Description Logic \( \mathcal{EL} \). In this DL, classes of the knowledge domain can be described by \( \mathcal{EL} \)-concepts, which are built from a set \( N_C \) of concept names and a set \( N_R \) of role names using the following rule:

\[
C, D ::= A | \top | C \sqcap D | \exists r.C
\]

where \( A \in N_C \), \( r \in N_R \) and \( C, D \) are concepts. General concept inclusions (GCI for short) of the form \( C \sqsubseteq D \) express that one concept \( C \) subsumes another concept \( D \), and are collected in a TBox. Additionally, assertion of the form \( C(a) \) and \( r(a, b) \) can express knowledge about individuals \( a, b \) from a set of individual names \( N_I \). An ABox is a set of assertions, while a knowledge base (KB) consists of both an ABox and TBox. The semantics of \( \mathcal{EL} \) are defined using the notion of interpretations \( I = (\Delta^I, \cdot^I) \) consisting of the interpretation domain \( \Delta^I \), and the interpretation function \( \cdot^I \). Each concept name is interpreted as a subset, each role name as a binary relation and each individual as an element of \( \Delta^I \). The interpretation is extended to \( \mathcal{EL} \)-concepts by interpreting \( \top^I \) as the full domain \( \Delta^I \), \( (C \sqcap D)^I \) as the intersection of \( C^I \) and \( D^I \), and existential restrictions \( \exists r.C \) as the image of \( C^I \) under \( r^I \). Then an interpretation satisfies a GCI \( C \sqsubseteq D \) if \( C^I \subseteq D^I \), and assertions \( C(a) \) and \( r(a, b) \) if \( a^I \in C^I \) and \( (a^I, b^I) \in r^I \), respectively. It satisfies a KB \( K = (T, A) \) if it satisfies all GCIs in the TBox \( T \) and all assertions in the ABox \( A \). The following are commonly used reasoning tasks: Concept subsumption \( C \sqsubseteq_T D \) asks, given a TBox \( T \) and two concepts \( C \) and \( D \), whether \( C \) is subsumed by \( D \) w.r.t. \( T \), i.e., \( C^I \subseteq D^I \) for all models \( I \) of \( T \). Given an individual \( a \), a concept \( C \), and a KB \( K \), \( a \) is called an instance of \( C \) w.r.t. \( K \), denoted \( K \models C(a) \), iff \( a^I \in C^I \) for all models \( I \) of \( K \).

Let \( \mathcal{C}(\mathcal{EL}) \) be the set of all \( \mathcal{EL} \)-concept descriptions. A concept similarity measure \( \sim_T \) w.r.t. a TBox \( T \) is a function \( \sim_T : \mathcal{C}(\mathcal{EL}) \times \mathcal{C}(\mathcal{EL}) \to [0, 1] \), where \( C \sim_T C = 1 \) for all concepts \( C \in \mathcal{C}(\mathcal{EL}) \). Several properties of CSMs have been formalized in [5], the most important ones here are symmetry and equivalence invariance; the latter expresses that the similarity value between two concepts remains the same when replacing one concept
for an equivalent one w.r.t. $\mathcal{T}$. Based on this notion we can formalize the central inference as follows:

**Definition 1** (relaxed instance). The individual $a$ is a *relaxed instance* of the query concept $Q$ w.r.t. the KB $\mathcal{K}$, the CSM $\sim_{\mathcal{T}}$ and the threshold $t \in [0, 1]$ iff there exists a concept description $X$ such that $Q \sim_{\mathcal{T}} X > t$ and $\mathcal{K} \models X(a)$.

To compute the relaxed instances of an $\mathcal{EL}$-concept (w.r.t. an $\mathcal{EL}$-KB) it is not feasible to compute all sufficiently similar concepts and then perform instance checking for those, since (1) the number of those concepts can be infinite leading to an infinite number of queries and (2) a similarity measure does not necessarily provide a method how to obtain a ‘sufficiently similar’ concept.

In [2], we showed how relaxed instance queries can be solved for unfoldable TBoxes, which basically only allow to introduce abbreviations for concepts and can be dropped after expanding all concepts. It also restricts to those similarity measures, that are equivalence invariant and work by recursively comparing the structure of the concepts. However, for general $\mathcal{EL}$-TBoxes as introduced above, the approach described in [2] fails, as the query concept may have a cyclic definition and thus can not be expanded. Additionally, we were not able to find a structural, equivalence invariant measure that works for concepts w.r.t. to a general TBox. To solve this, we introduce a CSM $\sim_c$ that uses the canonical models of a concept $C$ w.r.t. a TBox $\mathcal{T}$, denoted with $\mathcal{I}_{C,\mathcal{T}}$ [6], and a similarity measure $\sim_i$ between interpretations as follows: $C \sim_c D = (\mathcal{I}_{C,\mathcal{T}}, d_C) \sim_i (\mathcal{I}_{D,\mathcal{T}}, d_D)$, where $d_C$ and $d_D$ are the elements in their respective canonical models corresponding to the concepts $C$ and $D$.

The family of CSMs $\sim_c$ for $\mathcal{EL}$ inherits several formal properties from $\sim_i$, in particular symmetry and equivalence invariance. The interpretation similarity measure (ISM) $\sim_i$ can be parametrized by a weighting function that assigns different weights to each concept and role name, by a primitive measure between concept and role names and by a discounting factor. Using those parameters, it is possible to achieve the different weighting of features as described in Example 1 for $\sim_{\text{flying}}$ and $\sim_{\text{diet}}$. For relaxed instance queries, this means that those parameter together with the threshold $t$ allow to specify, which features may be relaxed, and which should stay close to the query concept.

The ISM $\sim_i$ between elements of two interpretations is defined recursively by comparing the concept names, that those elements are instances of, and the successors they have in the interpretation domain, i.e., roles that connect those elements to others (the complete definition can be found in [3]). If written as an equation system, solving this system will yield the similarity values between all pairs of elements of the two interpretations. Indeed, this equation system can be transformed into a linear optimization problem, and can be solved in polynomial time.

If we built such an equation system for the canonical model $\mathcal{I}_{Q,\mathcal{T}}$ of the query concept $Q$ and the canonical model $\mathcal{I}_{A,\mathcal{T}}$ of the ABox $\mathcal{A}$ that we want to query (both w.r.t. the TBox $\mathcal{T}$), and modify this equation system by guessing the best subsets of the concept names and role-successors for the elements in $\mathcal{I}_{A,\mathcal{T}}$, we actually get a nondeterministic algorithm that computes, to which degrees all of the individuals in the ABox $\mathcal{A}$ belong to the query concept $Q$ w.r.t. the CSM $\sim_c$ and the TBox $\mathcal{T}$. Then it is sufficient to check for each individual $a$, if the degree is larger than the threshold $t$. This algorithm is sound and complete, and allows us to formulate the following theorem:
Theorem 2. Relaxed instances of a query concept $Q$ w.r.t. $\sim_c$ and a general knowledge base $K$ is in NP.

To conclude, we have proposed a new reasoning service that allows relaxed instance query answering for application-specific notions of similarity by the appropriate choice of a CSM $\sim_T$ and threshold $t$. We have introduced a new family of CSMs that take the whole information from general TBoxes into account. The $\sim_c$ CSMs are, to the best of our knowledge, the first CSMs of this kind for general TBoxes. Based on these we gave a computation algorithm for relaxed instances w.r.t. general TBoxes. For more details see [3, 4] and for its extension to $\mathcal{EL}^{++}$ see [1].

References


Introduction  Formal verification methods have (with few exceptions) been developed for – and are thus tailored to – statically typed programming languages. Recently, dynamically-typed programming languages are growing in popularity even for business- and safety-critical applications. Also, the ubiquity of JavaScript as the “assembly language of the web” is widely acknowledged. The increasing need for safety guarantees for this kind of programs is echoed in the research community by an increasing number of proposals adapting verification methods from statically-typed to dynamically-typed programs [6, 9, 3, 8]. However, the absence of statically known type information in dynamically-typed programs usually creates significant overhead when applying such methods and significantly reduces the effectiveness of automated reasoning engines.

In [5], we investigated the impact of such adaptions by comparing an adapted Hoare logic [6] with its traditional statically-typed counterparts [1, 2] (see Section B). Based on these observations, [5] makes the following main contributions:

- A layer of abstraction, abstracting from the complexity of dynamic typing and thus allowing to verify dynamically-typed programs just like statically-typed ones given sufficient type information.

- A methodology for semi-automatically deriving such type information based on integrating a type analysis into Hoare logic and exchanging results bidirectionally. The methodology requires manual intervention only when the type analysis is insufficient and is complete relative to the Hoare logic used. It also allows interleaving proof steps for type safety with those for other properties.

To allow for properly discussing the generality of these results, we will now give a brief account of formal verification methods in general:

A. Formal Methods  Software Verification is the attempt to logically reason about program behavior. Formal Methods are distinguished from testing by the fact that they are not concerned with a particular program execution from a particular start state, but rather with all possible program executions from all possible (or reachable) start states. For such methods it is of vital importance to concisely represent (possibly infinite) sets
of states that a program execution (or part thereof) can start or result in. Depending on
the method’s focus, this can be achieved in different ways.

A.1. Focus on Automation  Formal methods focussing on automation are often called
“program analysis” or “automatic verifier” to emphasise their fully automatic nature.
They are mostly based on the concept of Abstract Interpretation [4] and usually use finite
abstract domains to ensure termination:

A (possibly infinite) value domain \( D \) is abstracted into a (usually finite) complete
lattice \( (A, \sqsubseteq) \) called the “abstract domain” using a Galois connection\(^1\). Examples include
type abstractions, parity abstractions, sign abstractions and interval abstractions. The
abstraction is then lifted to program states in the natural way, enabling the use of abstract
states like \( \{x \mapsto [1, 5], y \mapsto [2, 4]\} \) for denoting sets of concrete ones.

Program operations are also abstracted to work on abstract values instead of concrete
ones. For the statement \( r := x + y \), this allows deriving properties like:

\[
\{x \mapsto [1, 4], y \mapsto [5, 10]\} \quad r := x + y \quad \{x \mapsto [1, 4], y \mapsto [5, 10], r \mapsto [6, 14]\}
\]

Note that the abstract postcondition contains concrete states like \( \{x \mapsto 1, y \mapsto 5, r \mapsto 14\} \)
that cannot result from executing the above statement from any start state. Such over-
approximation is necessary since the abstract domain of intervals is not flexible enough
to precisely express the real set of result states.

In essence, such approaches simulate (“interpret”) runs of abstracted programs on
abstracted data, iterating loops and recursion until reaching a fixpoint. With all transfer
functions monotone and the abstract domain finite, Tarski’s fixpoint theorem guarantees
termination. In case of infinite abstract domains, a technique called “widening” acceler-
ates the process by enforcing larger over-approximation steps at the expense of precision.

A.2. Focus on Completeness  Formal Methods focussing on completeness are often
called “program logics”. They avoid the loss of precision outlined above by using ex-
tremely flexible formula-based state set representations. Similar to what mathematicians
do when using variables representing arbitrary values instead of concrete values, symbolic
variables \( v_{u_0} \) are used to represent the initial values of variables \( u \) and all further values
are represented as terms over these variables. Such methods are therefore often called
“symbolic”. Applying them to above example yields the following property:

\[
\{x \mapsto v_{x_0}, y \mapsto v_{y_0}\} \quad r := x + y \quad \{x \mapsto v_{x_0}, y \mapsto v_{y_0}, r \mapsto v_{x_0} + v_{y_0}\}
\]

which is obviously more precise than the postcondition derived by abstract interpreta-
tion above. Note that such term-based or formula-based representations can be seen as an
(infinite) abstract domain. In fact, applying abstract interpretation using such a “sym-
bolic” domain would be very similar to approaches like symbolic execution or Dijkstra’s
strongest-postcondition calculus. Also, such approaches are often parameterized in the
logic, since it’s expressivity correlates to the flexibility of the abstract domain.

While increasing precision is clearly preferable, these formal methods usually require
manual assistance to deal with loops and recursion as naively applying them would iterate

\[^1\text{a pair of a monotone abstraction function } \alpha : 2^D \mapsto A \text{ and a monotone concretization function } \gamma : A \mapsto 2^D, \text{ such that } \alpha \circ \gamma \subseteq id_A \text{ and } \gamma \circ \alpha \subseteq id_D\]

Frontiers of Formal Methods 2015
such loopy control flows indefinitely, creating larger and larger symbolic state representations. Verifiers based on such approaches hence ask the user to specify loop invariants and method contracts a priori and apply automatic reasoning only along straight-line program segments to check these user-supplied assertions for mutual consistency.

A.3. Completeness vs. Automation While precision is necessary for completeness, termination is necessary for automation. The over-approximation in abstraction-based approaches enables termination at the expense of precision and hence aims at automation. The symbolic representation of state sets in formula-based approaches increases precision at the expense of termination and hence aims at completeness.

B. Static vs. Dynamic Typing In [5], we compared an (adapted) Hoare logic for JavaScript [6] with its more traditional counterparts [1, 2] for statically-typed languages. We were able to identify the following key sources for additional complexity in verifying dynamically-typed programs. To show that our observations are not particular to Hoare logic, but stem from its formula-based nature, we will here offer a generalized explanation:

B.1. Mapping Predicates Formula-based formal methods rely on their ability to symbolically represent the results of programming-language operations as equivalent terms over symbolic values. However, in statically-typed programs the well-typedness of these terms rests on the following assumptions:

- logic and programming language must share the same type system,
- all programming language operations must also be available in the logic,
- each symbolic value \( v_{u_0} \) must have the same type as the variable \( u \) whose value it represents, and
- the program must be well-typed.

While these assumptions are often left implicit, relinquishing any one of them opens the door for potential logical inconsistencies as the usual two-valued logic does not leave room for type-errors. To reuse the example from above, if the variable \( x \) would reference a boolean value instead of an integer, the term \( v_{x_0} + v_{y_0} \) would not make sense as the operation + is undefined for boolean values. Usually, this is mitigated by treating program variables as untyped and introducing mapping predicates for mapping between such untyped program values and their respective typed logical values.

B.2. Pure Expressions When all operations and types of the programming language are also available in the assertion language, then the symbolic representation of a program expression’s result nearly always looks exactly like this very program expression - just with variables \( u \) substituted by their respective symbolic values \( (v_{u_0}) \). From this observation, it usually does not take long until someone introduces shortcut rules – proof

\[ ^2 \text{just like in abstract interpretation with an infinite abstract domain without widening} \]
\[ ^3 \text{in object-oriented languages usually all variables are given the type } \text{O} \text{ of objects} \]
\[ ^4 \text{like the predicates True and False in [6], which are similar to our } B(\_, \text{true}) \text{ and } B(\_, \text{false}). \]
rules that allow transferring program expressions directly into assertions instead of step-wise dissecting their syntax-tree while reconstructing their logical representation in the assertion language. These two classic Hoare logic rules are examples for such shortcuts:

\[
\begin{align*}
\{p[u := e]\}u := e\{p\} & \quad \{p \land b\}S_1\{q\} & \quad \{p \land \neg b\}S_2\{q\} \\
\{p\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \text{ end } \{q\}
\end{align*}
\]

B.3. Type Safety Preconditions In dynamically-typed programs, type errors are runtime events and hence need to be avoided just like failure or divergence. Proof rules for type-safe notions of correctness hence include type-safety preconditions – additional requirements for safeguarding program executions against type errors. In statically-typed programs, these are not necessary.

C. Layer of Abstraction In [5], we could show that, given sufficient type information, it is possible to mitigate all these sources of complexity and verify dynamically-typed programs just like statically-typed ones – that is, with all conveniences that statically known type information enables: objects may be automatically mapped into typed values, type-safety preconditions may be omitted and shortcut rules allowed. However, deriving such type information for a dynamically-typed program is itself an uncomputable problem.

C.1. Type Information in its Various Forms Although uncomputable, we noticed that type information can be extracted from type safety proofs in Hoare logic. Since such proofs (like with other symbolic approaches) are derivable for all type-safe programs due to the (relative) completeness of Hoare logic, it follows that the needed type information exists (and can be derived) for all dynamically-typed programs that are type-safe. In [5] we could show the entire procedure to be complete relative to the Hoare logic used.

D. Semi-Automatically Deriving Type Information Since manually deriving type safety proofs is tedious, we integrated a type inference algorithm based on data-flow analysis (an abstraction-based program analysis) into our program logic with the aim of automating as much of the process as possible. The type inference algorithm first derives a basic typing for the entire program that is used to support manual Hoare logic proofs as outlined above. Whenever type information can be extracted from a user’s manual proof, it is used as a trusted assumption to increase the analysis’s precision. Such refinements do not break any previously derived guarantees\(^5\), thus turning verification into an iterative back-and-forth between automated analysis and manual proving.

Discussion Formula- and abstraction-based formal methods have complementary strength and weaknesses. Several researchers have proposed different ways of combining them [8, 7].

Nguyen et al. [8] propose a contract verifier based on symbolic execution, but use a technique similar to widening to enforce termination. This mechanism can be seen as a form of abstraction (and also causes a loss of precision). Their approach hence combines symbolic with abstraction-based reasoning.

Khoo et al. [7] propose a framework for integrating symbolic execution with type analysis. First, the user partitions his/her program into so-called s- and t-blocks. The

\(^5\)Due to a monotonicity property inherent in data-flow analyses
Formally Verifying Dynamically-typed Programs like Statically-typed Ones  
A different perspective

analysis then applies type analysis to t-blocks while using (exhaustive and hence sound) symbolic execution on s-blocks. The derived results are bidirectionally exchanged using MIX rules. In essence, analysis results are used to construct start states for symbolic execution and types derived by symbolic execution can be used in the type analysis. Our proposal for semi-automatically deriving type information ([5, Section 6], Section D) also falls into this category as it combines a formula-based program logic with an automated, abstraction-based type inference.

Since nothing about our approach is particular to Hoare logic, it is reasonable to believe that a similar construction is possible with other formula-based approaches to program verification like Floyd’s inductive assertions, symbolic execution, refinement types or Dijkstra’s predicate transformer semantics.

As detailed in Section B, the layer of abstraction can also be expected to apply to formula-based approaches in general. However, it requires type information and is hence tied to type analysis. Nevertheless, the entire concept seems applicable to arbitrary formula-based formal methods for dynamically-typed programming languages, complementing them with an abstraction-based type-analysis to decrease the annotation burden and increase the effectiveness of automated reasoning engines as was the aim of [5].

References

Path-Checking for MTL and TPTL over Data Words

Shiguang Feng†(shiguang.feng@informatik.uni-leipzig.de)
Institut für Informatik, Universität Leipzig, Germany

1 Introduction

Linear time temporal logic (LTL) is nowadays one of the main logical formalisms used for the specification and verification of reactive systems, and has found applications in industrial tools. Two extensions of LTL are MTL (metric temporal logic) [5] and TPTL (timed propositional temporal logic) [1]. In MTL, the temporal operators next (X) and until (U) are indexed by time intervals. For instance, the formula \( p U_{[2,3)} q \) holds at a certain time \( t \), if there is a time \( t' \in [t+2, t+3) \), where \( q \) holds, and \( p \) holds during the interval \([t, t')\). TPTL is a more powerful logic that is equipped with a freeze formalism. It uses register variables, which can be set to the current time value and later these register variables can be compared with the current time value. Each MTL-formula is equivalent to a TPTL-formula. For instance, the above MTL-formula \( p U_{[2,3)} q \) is equivalent to the TPTL-formula \( x. (p U (q \land 2 \leq x < 3)) \). Here, the constraint \( 2 \leq x < 3 \) should be read as: The difference of the current time value and the value stored in \( x \) is in the interval \([2, 3)\). We consider the model-checking problem of MTL and TPTL over finite or infinite data words. Let \( P \) be a finite set of atomic propositions. A data word over \( P \) is a finite or infinite sequence \((P_0, d_0)(P_1, d_1) \cdots \) of pairs from \( 2^P \times \mathbb{N} \). It is monotonic, if \( d_i \leq d_{i+1} \) for all appropriate \( i \). It is pure, if \( P_i = \emptyset \) for all \( i \geq 0 \).

As for TPTL, freeze LTL can store the current data value in a register \( x \). But in contrast to TPTL, the value of \( x \) can only be compared for equality with the current data value. For model-checking freeze LTL the authors of [3] consider one-counter machines (OCM) as a mechanism for generating infinite non-monotonic data words, where the data values are the counter values along the unique computation path. Whereas freeze LTL model-checking for non-deterministic OCM turned out to be \( \Sigma^1_1 \)-complete, the problem becomes \( \text{PSPACE} \)-complete for deterministic OCM [3].

We extend the \( \text{PSPACE} \)-completeness result for freeze LTL to TPTL over non-monotonic data words. Our first main result states that model-checking for TPTL over deterministic OCM is \( \text{PSPACE} \)-complete. This sharpens the decidability result from [6] and at the same time generalizes the \( \text{PSPACE} \)-completeness result for freeze LTL. We also show that \( \text{PSPACE} \)-hardness already holds (i) for the fragment of TPTL with only two register variables and (ii) for full TPTL, where all interval borders are encoded in unary (the latter result can be shown by a straightforward adaptation of the \( \text{PSPACE} \)-hardness proof in

*This is a joint work with Markus Lohrey and Karin Quaas.
†The author is supported by the DFG Research Training Group 1763.
On the other hand, if we restrict TPTL to (i) a constant number of register variables and unary encoded numbers in constraints, or (ii) one register variable but allow binary encoded numbers, then model-checking over deterministic OCM is P-complete. Note that the latter covers MTL over non-monotonic data words.

We actually prove our upper complexity bounds for infinite periodic data words. Such a data word is given by two finite data words $u = (P_1, d_1) \cdots (P_m, d_m)$ and $v = (Q_1, e_1) \cdots (Q_n, e_n)$ (where $d_1, \ldots, d_m, e_1, \ldots, e_n \in \mathbb{N}$) together with an offset number $K$. The resulting infinite data word is $u \prod_{i \geq 0} (v + iK)$, where $v + M$ denotes the data word $(Q_1, e_1 + M) \cdots (Q_n, e_n + M)$. It can be easily seen that the infinite data word produced by a deterministic OCM is such a periodic data word.

We write $\text{TPTL}_{\text{u}}$, $\text{TPTL}_{\text{u}}$ and $\text{MTL}_{\text{u}}$ (resp. $\text{TPTL}_{\text{b}}$, $\text{TPTL}_{\text{b}}$, and $\text{MTL}_{\text{b}}$) if we want to emphasize that numbers are encoded in unary (resp., binary) notation.

## 2 Lower Bound

**Theorem 1.** Path-checking for $\text{TPTL}_{\text{u}}^1$ over finite unary encoded strictly monotonic pure data words is P-hard.

Proof Sketch. A synchronous alternating monotone circuit with fanin 2 and fanout 2 (SAM2-circuit) is a monotone variable-free circuit divided into levels such that all gates in the same level are of the same type and the levels alternate between $\land$-levels and $\lor$-levels. The circuit value problem for SAM2-circuits (SAM2CVP) is P-complete [4]. We reduce from SAM2CVP to the path-checking problem for $\text{TPTL}_{\text{u}}^1$. □

**Theorem 2.** Path-checking for $\text{TPTL}_{\text{u}}$ over finite unary encoded strictly monotonic pure data words is PSPACE-hard.

Proof Sketch. We show PSPACE-hardness by a reduction from the PSPACE-complete quantified Boolean formula problem (QBF). □

The constructions in the proof of Theorem 1 and 2 can be easily adapted to infinite data words. The next lower bound only holds for infinite data words.

**Theorem 3.** Path-checking for $\text{TPTL}_{\text{b}}^2$ over infinite strictly monotonic pure data words is PSPACE-hard.

Proof Sketch. The theorem is proved by a reduction from the PSPACE-complete quantified subset sum problem (QSS), see [7]. □

## 3 Upper Bound

**Theorem 4.** Path-checking for $\text{TPTL}_{\text{b}}$ over infinite binary encoded data words is in PSPACE.

Proof Sketch. Fix two finite data words $u_1, u_2$, a number $k \in \mathbb{N}$ and a TPTL-formula $\psi$, and let $w = u_1 \prod_{i \geq 0} (u_2 + ik)$. We show that one can decide in APTIME = PSPACE whether $w \models \psi$ holds. □

If all numbers are unary encoded and the number of register variables is fixed, then the alternating Turing-machine from the proof of Theorem 4 works in logarithmic space. Since ALOGSPACE = P, we obtain:
Theorem 5. For every fixed \( r \in \mathbb{N} \), path-checking for \( \text{TPTL}^r_u \) over infinite unary encoded data words is in \( P \).

For finite data words, there is a polynomial time algorithm also for binary encoded data words (assuming again a fixed number of register variables). The point is that we only have to consider polynomially many register valuations.

Theorem 6. For every fixed \( r \in \mathbb{N} \), path-checking for \( \text{TPTL}^r_b \) over finite binary encoded data words is in \( P \).

For infinite data words we have to reduce the number of register variables to one in order to get a polynomial time complexity for binary encoded numbers:

Theorem 7. Path-checking for \( \text{TPTL}^1_b \) over infinite binary encoded data words is in \( P \).

Figure 1 collects our complexity results for path checking problems. The lower bounds all hold for pure strictly monotonic unary encoded data words. The upper bound hold for general (non-pure and non-monotonic) data words that are binary encoded, except for \( \text{TPTL}^{< \infty}_u \) (membership in \( P \)), where the data word has to be unary encoded.

For a given DOCM \( A \) one can check in logspace, whether \( \text{run}(A) \) is finite or infinite. Moreover, if \( \text{run}(A) \) is finite, then the corresponding data word in unary encoding can be computed in logspace. If \( \text{run}(A) \) is infinite, then one can compute in logspace two unary encoded data words \( u_1 \) and \( u_2 \) and a unary encoded number \( k \) such that \( \text{run}(A) = u_1 \prod_{i \geq 0} (u_2 + ik) \) [3]. The diagram from Figure 1 also shows the complexity results for TPTL-model-checking over DOCM.

References


Threshold Concepts in a Lightweight Description Logic*

Oliver Fernández Gil (fernandez@informatik.uni-leipzig.de)

Department of Computer Science, University of Leipzig, Germany

Description Logics (DLs) [1] are a well-investigated family of logic-based knowledge representation formalisms. They can be used to represent the relevant concepts of an application domain using concept descriptions, which are built from sets of concept names and role names using certain concept constructors. Using concept descriptions one can define important concepts of interest, by expressing the necessary and sufficient conditions for an individual to belong to the concept. The semantics for these concepts is given by usual first-order logic interpretations where a non-empty domain of elements is assumed, concept names are interpreted as subsets from the domain and role names as binary relations over the domain.

In particular the lightweight DL $\mathcal{EL}$, which offers the concept constructors conjunction ($\sqcap$), existential restriction ($\exists r.C$) and the top concept ($\top$), has shown to be of great interest. On the one hand, though quite inexpressive, it can be used to define large biomedical ontologies such as SNOMED CT$^1$. On the other hand, important inference problems like subsumption have been shown to be decidable in polynomial time, even with respect to terminological axioms [2]. In $\mathcal{EL}$ we can, for example, define the concept of a happy man as a male human that is healthy and handsome, has a rich and intelligent wife, a son and a daughter, and a friend:

\begin{equation}
\text{Human} \sqcap \text{Male} \sqcap \text{Healthy} \sqcap \text{Handsome} \sqcap 
\exists \text{spouse}.(\text{Rich} \sqcap \text{Intelligent} \sqcap \text{Female}) \sqcap 
\exists \text{child}.\text{Male} \sqcap \exists \text{child}.\text{Female} \sqcap \exists \text{friend}.\top
\end{equation}

Since concept descriptions are interpreted using the classical semantics from first-order logic, for an individual to belong to this concept, it has to satisfy all the stated properties. However, maybe we would still want to call a man happy if most, though not all, of the properties hold. It might be sufficient to have just a daughter without a son, or a wife that is only intelligent but not rich, or maybe an intelligent and rich spouse of a different gender. But still, not too many of the properties should be violated.

This work presents a DL extending $\mathcal{EL}$ that allows to define concepts in such an approximate way. The main idea is to use a membership degree function, which instead of giving a value in $\{0, 1\}$ to evaluate the membership of an individual into a concept, gives a value in the interval $[0..1]$. Then, from an $\mathcal{EL}$ concept description $C$, we can build the threshold concept $C_{\geq t}$ for $t \in [0..1]$ which contains all the individuals that belong to

*Supported by DFG Graduiertenkolleg 1763 (QuantLA).

$^1$see http://www.ihtsdo.org/snomed-ct/
C with degree at least \( t \). In addition, we also allow the construction of lower threshold concepts of the form \( C_{\leq t} \) and the use of strict inequalities. Based on this, for instance, we can then require a happy man to belong to the \( \mathcal{EL} \) concept (1) with degree at least \( .8 \). Conversely, an unhappy man could be required to belong to the \( \mathcal{EL} \) concept (1) with degree less than \( .2 \).

We extend \( \mathcal{EL} \) by adding new threshold concept constructors which are based on an arbitrary, but fixed graded membership function satisfying certain minimal requirements. Then, we introduce a specific graded function which satisfies these requirements. Its definition is a natural extension of the homomorphism characterization of crisp membership in \( \mathcal{EL} \). More precisely, in \( \mathcal{EL} \) concept descriptions and interpretations can be represented by \( \mathcal{EL} \) description trees and \( \mathcal{EL} \) description graphs, respectively [3, 4]. Then, membership in \( \mathcal{EL} \) can be characterized via existence of homomorphisms between \( \mathcal{EL} \) description graphs[3].

The general idea is the following: given an interpretation \( \mathcal{I} \), a concept \( C \) and an individual \( d \) from the domain of \( \mathcal{I} \); the degree of membership of \( d \) in \( C \) decreases, in a uniform way, for each property of \( C \) that \( d \) does not have. To identify these missing properties we explore all the possible ways in which the structure of \( C \) can be (partially) mapped into the structure of \( d \). Further, we define a weighted function which associates to each partial mapping a corresponding value in the interval \([0..1]\), that tells how far away is \( d \) from \( C \) according to the mapping. The maximum among these values is then taken as the degree of membership for \( d \) in \( C \).

Finally, we investigate the computational properties of extending \( \mathcal{EL} \) with threshold concepts using our specific membership function. Basically, we look at standard reasoning tasks in DLs, e.g., concept satisfiability, subsumption, ABox consistency and instance checking. We obtain NP-completeness for satisfiability and consistency; and coNP-completeness for subsumption. Regarding instance checking we consider the two well-known criteria of data and combined complexity. For data complexity the instance problem is coNP-complete, whereas a preliminary coNEXP-time upper bound is obtained with respect to combined complexity.

This is a joint work with Franz Baader and Gerhard Brewka.

References


Graph grammars have been extensively studied, mainly to characterise the sets of graphs generated by them [3, 6, 4]. It is well known that arbitrary Turing machines can be simulated with graph grammars. However, despite the computational power of graph grammars the language of control expressions is a decidable set of strings for any given graph grammar. By this we mean the rule label sequences of possible derivations from a given start graph. Graph transformation as a string language generation device is obviously very powerful and includes the corresponding notion for Chomsky grammars, which is known as Szilard languages. It also generalizes Petri net languages.

Double-pushout graph transformation rules specify a left hand side $L$, a right hand side $R$ and a common subgraph of both, the interface $K$. Identifying $L$ as a subgraph of the graph to be transformed induces a transformation step, provided that removing the items of $L - K$ still results in a graph (no node may be deleted unless all edges incident to it are also deleted). The successor graph is then uniquely determined by removing the nodes and edges of $L - K$ and adding those of $R - K$. Figure 1 shows a few derivation steps. Each step corresponds to a rule application, and the images of the left hand sides under the respective matches have been highlighted. The forms of the rules can be deduced because the rule effect is always a local change in the graph, consuming and producing a fixed pattern of nodes and edges.

Our motivation is to formulate language-theoretic correctness problems for systems that are modeled by graph transformation rules: given a graph grammar, is there an execution sequence outside of the specification language? There exists some recent work where rewriting games on finite structures, including graphs, are solved [5] which we take as evidence for the relevance of such questions. This abstract is about our paper [2], and some supplementary, as yet unreviewed, results found since its publication.

In [2], we defined $\mathcal{L}_S$ as the family of terminal derivation languages of double-pushout graph grammars, where terminal states are recognised by a graph rule, equivalently a finite set of such rules. The central result in that paper was the closure of $\mathcal{L}_S$ under (nondeleting) letter-to-letter homomorphisms, which is nontrivial because rules are labeled unambiguously in our formalism.
Theorem 1. If $L \in \Sigma^*$ is the terminal derivation language of a graph grammar $G$ and $h : \Sigma^* \to \Sigma^*$ is a letter-to-letter homomorphism, then $h(L)$ is the terminal derivation language of another graph grammar $G'$.

Proof idea. By induction over $|\text{dom}(h) - \text{cod}(h)|$, $h$ is decomposed into a finite number of two-letter mergers. The two-letter mergers are handled by constructing a single rule out of the rules whose labels are merged, and extending the start state as sketched below:

![Diagram of graph grammar construction](image)

$G$ is the original start graph, $L_1$ and $L_2$ are the left hand sides of the two rules to be merged, respectively. The nodes and edges in the top serve to guide the simulation: every node originally present, or created during the derivation, is additionally linked to one of the nodes 0, 1, 2 or 3 via an extra edge. The combined rule is crafted in such a way that a match is composed of a match of $L_1$ and a match of $L_2$, respecting the guide nodes and edges. When the first rule is simulated, the match of $L_1$ consists entirely of nodes linked to 0 (and edges between these) and the match of $L_2$ consists entirely of nodes all linked to $i$, where $i$ is either 1 or 2. Node 3 − $i$ is then refurbished with a new $L_2$, and node 3 with a new copy of $L_1$ (adding a new $L_2$ is necessary so we can always simulate the first rule again; adding a new $L_1$ is unavoidable because the rule effect, in terms of what is deleted and added, is fixed). The situation when the second rule is simulated is analogous. Node 3 is entirely for garbage – surplus $L_1$’s and $L_2$’s accumulated because the simulation of choice between the two original rules must really always effect both. The construction is proven correct by an induction over the length of the derivation, where it is shown that (a) the simulation can always proceed and (b) no spurious steps are created.

The closure property is very useful for showing the containment of other families in $\mathcal{L}_5$. As a basic example, we showed that the context-free languages are in $\mathcal{L}_5$ (immediately obtaining undecidability of many questions like emptiness because $\mathcal{L}_5$ is also closed under intersections). There are more extensive known families which are also contained, for instance we could subsequently show that the growing context-sensitive languages (GCSL) [1] are terminal derivation languages of graph grammars. The proof makes heavy use of the nondeterminism inherent in graph grammars and is aided by the fact that any finite sequence of graph transformation rules can be replaced by a finite set of graph transformation rules (not sequences) with the same effect. This allows a graph grammar to perform a simulation of the derivation of a word $w$ in the GCS grammar in $\lceil \frac{|w|}{2} \rceil$ steps, while the remaining steps are used for checking the consistency of the simulation and whether the rules used (whose labels are recorded in another string-like part of the graph) spell out $w$, and enabling the end rule only if both conditions are met.

Unsurprisingly, $\mathcal{L}_5$ does appear to be very large and we could not ascertain whether all of the languages in our family are even context sensitive (the straightforward simulation by a nondeterministic Turing machine uses a slightly more than linear amount of

---

Frontiers of Formal Methods 2015

100
space). Conversely, an open conjecture is that the language of \( \{ a^p \mid p \text{ prime} \} \) is not in \( \mathcal{L}_s \). Extending our work published in [2], we found it useful to restrict attention to graph grammars that only produce graphs of bounded pathwidth. Although it is not a syntactic restriction and the restriction itself is undecidable, multiple advantages would be gained (the following results being preliminary and not yet published): all languages obtained thus are indeed context sensitive; it seems that the closure properties we found (\( \mathcal{L}_s \) being in fact an intersection-closed AFL, i.e. closed under intersection, union, concatenation, Kleene plus, non-deleting homomorphisms, inverse homomorphisms and intersections with regular languages) are not destroyed, because the constructions only increase the pathwidth bound; furthermore, the more pathological examples seem to be excluded. We are aware that in its current form, our work does not solve the problem of finding classes of graph grammars that allow verification of language-theoretic correctness, for example compliance with a specification in linear temporal logic. A search for practically relevant restricted versions of \( \mathcal{L}_s \) could move us closer to that goal.

References


Decision Procedure for Stochastic Satisfiability Modulo Theories with Continuous Domain*  

Yang Gao (yang.gao@uni-oldenburg.de)  

SCARE Research Training Group, Oldenburg University, Germany

1 Introduction

The idea of modelling uncertainty within propositional satisfiability (SAT) was first introduced in [1] by adding randomized quantifiers to the SAT formula. The resulted formula is an SAT formula bounded by a sequence of quantifiers, i.e., an SSAT formula with classic quantifiers and randomized quantifiers. This work has been extended by Tino Teige et al [2, 3], in their innovative work, the randomized quantifiers are introduced to SMT (Satisfiable Modulo Theory) so that a larger class of properties can be formulated and analysed. Instead of reporting true or false, an SSMT (or SSAT) formula \( \Phi \) will return a quantitative number, which is the maximum probability of satisfaction of \( \Phi \). Benefit from the expressive power of SSMT and SSAT, the concise description of diverse problems combining reasoning under uncertainty with data dependencies are permitted, which have been extensively studied in Artificial Intelligence [4, 5, 6], Probabilistic Hybrid Systems [2] etc.

The limitation of Tino Teige’s work is: the variables in an SSMT formula \( \Phi \) which bounded by randomized quantifiers can only take values in discrete domains, that is to say, SSMT formula considered in their work can only handle problems with discrete probability distributions. Due to the lack of continuity, a large number of properties can not be dealt with the SSMT/SSAT framework, such as the white noise in the system etc. Inspired by their work and in order to fill in the gaps between continuous and discrete probabilistic behaviors, we relax the constraints for the randomized variables and they can be assigned according to probability distributions, which can be either discrete or continuous.

2 Stochastic Satisfiability Modulo Theories with Continuous Domain

Definition 2.1. An SSMT formula with continuous domain (SSMT-C) is of the form: \( \Phi = Q : \varphi \), where:

*This research is funded by the German Research Foundation through the Research Training Group DFG-GRK 1765: “System Correctness under Adverse Conditions” (SCARE, scare.uni-oldenburg.de).
- $Q = Q_1x_1 \in \text{dom}(x_1) \ldots Q_nx_n \in \text{dom}(x_n)$ is a sequence of quantified variables, $\text{dom}(x_i)$ denotes the domain of variable $x_i$, $Q_i$ is either an existential quantifier $\exists$ or a randomized quantifier $\forall_i$, with probability density function $\pi_i$ satisfying $\int_{\text{dom}(x_i)} \pi_i(x_i)dx_i = 1$.

- $\varphi$ is an SMT formula over quantifier-free non-linear arithmetic theory $\mathcal{T}$. Without loss of generality, we assume that $\varphi$ is in conjunctive normal form (CNF), i.e., $\varphi$ is a conjunction of clauses, and a clause is a disjunction of (atomic) arithmetic predicates with the form $x_i$ or $\neg x_i$. $\varphi$ is also called the matrix.

**Definition 2.2.** The semantics of a SSMT-C formula $\Phi = Q : \varphi$ is defined by the maximum probability of satisfaction $Pr(\Phi)$ as follows:

- $Pr(\varphi) = 0$ if $\varphi$ is unsatisfiable.
- $Pr(\varphi) = 1$ if $\varphi$ is satisfiable.
- $Pr(\exists x_i \in \text{dom}(x_i) \ldots Q_nx_n \in \text{dom}(x_n) : \varphi) = \sup_{v \in \text{dom}(x_i)} Pr(Q_{i+1}x_{i+1} \in \text{dom}(x_{i+1}) \ldots Q_nx_n \in \text{dom}(x_n) : \varphi[v/x_i])$.
- $Pr(\forall x_i \in \text{dom}(x_i) \ldots Q_nx_n \in \text{dom}(x_n) : \varphi) = \int_{v \in \text{dom}(x_i)} Pr(Q_{i+1}x_{i+1} \in \text{dom}(x_{i+1}) \ldots Q_nx_n \in \text{dom}(x_n) : \varphi[v/x_i])\pi_i(v)dv$.

Figure 1 shows an example, which uses a semantic tree to indicate the probability of satisfaction w.r.t. an SSMT-C formula $\Phi$.

$$\Phi = \exists x \in [-1, 1]\mathcal{N}(0, 1)y \in (-\infty, +\infty) : (x > 0 \lor 2a \cdot \sin(4b) \geq 3) \land (y > 0 \lor 2a \cdot \sin(4b) < 1)$$

$$Pr(\Phi) = \max(0, 5, 1) = 1$$

![Figure 1: Semantics of a C-SSMT formula depicted as a tree.](image)

### 3 Decision Procedure for SSMT-C

#### 3.1 Decision Procedure

In this section, we will concentrate on the following decision problem:
Decision Procedure for Stochastic Satisfiability Modulo Theories with Continuous Domain

Given an SSMT-C formula $\Phi = Q_1 x_1 \in \text{dom}(x_1) \ldots Q_n x_n \in \text{dom}(x_n) : \phi$ and a reference probability $\delta$, the decision procedure will return:

- GE: if the lower bound of $Pr(\Phi)$ is greater than or equal to $\delta$;
- LE: if the upper bound of $Pr(\Phi)$ is less than or equal to $\delta$;
- Inconclusive: if $\delta$ is in the range of the bounds of $Pr(\Phi)$ w.r.t. the computation accuracy $\epsilon$.

The algorithm is equipped with the following structures:

- $C$: a set collecting the constraints which must be satisfied, initially assigned to $\emptyset$.
- $\rho$: an ordered list (corresponding to the order of variables in $Q$) which records the interval valuation for each variable, initially assigned to $[\text{dom}(x_1), \ldots, \text{dom}(x_n)]$.
- $H$: a set of computation cells, initially assigned to $\emptyset$.

### 3.1.1 SMT Level.

Rule (INI) adds the first computation cell to $H$, which contains:

1) the formula $Q : \Phi$ which to be decided;
2) domain for each variable, recorded in $\rho$;
3) the constraints $C$ which must be satisfied, initially it’s empty;
4) $(p, q)_i$ is an over approximation which ignores the variables in front of $x_i$ in $Q$, i.e., the probability estimation for $x_i$ and its substructure.

$$H \rightarrow H \cup \{(Q : \phi, \rho, C)^{(0,1)}\}$$ (INI)

If all the parts except one in one clause can not be satisfied w.r.t the current evaluation $\rho$, then this unit must hold. Rule (UP) corresponds to the unit propagation in classical DPLL framework.

$$L \lor l' \in \phi, \rho \not\models L$$

$$H' \cup \{(Q : \phi, \rho, C)^{(p,q)}\} \rightarrow H' \cup \{(Q : \phi, \rho, C \cdot \langle l' \rangle)^{(p,q)}\}$$ (UP)

If the range of a variable $x_j$ can be narrowed according to the constraints $C$ and current evaluation $\rho$, we update the evaluation set and the probability estimation. This can be done by performing interval constraint propagation rule (ICP).

$$\rho \xrightarrow{C} (x_j \sim b), \rho \not\models he (x_j \sim b)$$

$$H' \cup \{(Q : \Phi, \rho, C)^{(p,q)}\} \rightarrow H' \cup \{(Q : \Phi, \text{update}_\rho(x_j \sim b), C)^{\text{renewal}_\rho(p,q)}\}$$ (ICP)

where

$$\text{update}_\rho(x_j \sim b)(x_i) = \begin{cases} \rho(x_j) \cap \{x | x \sim b\}, & \text{if } x_i = x_j \\ \rho(x_j), & \text{otherwise} \end{cases}$$

and

$$\text{renewal}_\rho(p, q)_i = \begin{cases} (p, q)_i, & \text{if } x_j < x_i \\ \mathbb{P}(\rho(x_i) \times \cdots \times \rho(x_j) \cap \{x | x \sim b\} \times \cdots \rho(x_n))_i, & \text{otherwise} \end{cases}$$

When we are at a point that nothing can be done, i.e., $\Phi$ is inconclusive under current configuration, we perform the splitting rule (SPL) to split the current computation cell into two cells and update $\rho$ and probability estimation accordingly.
3.1.2 Constraint Solving Level.

When a conflict is obtained, i.e., the current evaluation $\rho$ and constraints $C$ lead to some empty sets, the computation cell can be marked with probability 0. As shown in rule (CFL).

\[
\rho \triangleright (x_i = \emptyset)
\]

(\text{CFL})

If the current evaluation $\rho$ is hull consistent w.r.t. the constraint set $C$, a constraint solving procedure can be performed in order to generate the inner approximation and outer approximation for the solution. Generally speaking, we will get some inner boxes and outer boxes. For inner boxes, we construct computation cells with real probability; for outer boxes, we construct computation cells with over approximation, as shown in rule (CNSIS).

\[
\rho \triangleright_{hc} C
\]

(CNSIS)

3.1.3 Stochastic SMT Level.

If two computation cells are combinative, i.e., the probability are estimated w.r.t. the same variable $x_i$. If $x_i$ is bounded by $\exists$, the two cells can be combined by adding the probability (Rule $\exists$-COM)); otherwise if it’s bounded by $\forall$, the two cells can be combined by adding the probability (Rule $\forall$-COM).

\[
\rho_{x_i \cup x_i^2} \text{ is the convex hull of } x_i^1 \text{ and } x_i^2
\]

($\exists$-COM)

\[
\rho_{x_i \cup x_i^2} \text{ is the convex hull of } x_i^1 \text{ and } x_i^2
\]

($\forall$-COM)

If all the computation cells in the same level have been tackled, the probability should be propagated to the upper level, the Rule (LFT) checks all the computation cells in $H$, and lift the cell to its upper level if all its siblings have been combined.

\[
\forall (Q : \phi, \rho, C) \in H : j \neq i \rightarrow H' \cup \{(Q : \phi, \rho, C)^{(p,q)} \}
\]

(LFT)

3.1.4 Termination.

At any time, if the estimate probability at $x_i$ is less equal than the reference probability $\delta$, the original formula is concluded to be $\delta$-LE, i.e., rule (LE); on the other hand, $\delta$-GE if the estimate probability is greater equal than $\delta$, i.e., rule (GE).

\[
q \leq \delta
\]

(LE)
Decision Procedure for Stochastic Satisfiability Modulo Theories with Continuous Domain

\[ p \geq \delta \quad (GE) \]

\[ H' \cup \{(Q : \phi, \rho, C)(p,q)\} \rightarrow GE\]

Otherwise, the formula is inconclusive w.r.t. \( \delta \) if the \( \delta \)-GE or \( \delta \)-LE can not be judged under the accuracy \( \epsilon \):

\[ q > \delta \land p < \delta, |p - q| < \epsilon \]

\[ H' \cup \{(Q : \phi, \rho, C)(p,q)\} \rightarrow \text{INCON} \]  

\[ (\text{INCON}) \]

4 Conclusion

In this short abstract, we briefly introduce our recent work on decision procedure w.r.t. stochastic SMT formula with continuous domain, which can be applied to reason and verify the systems with probabilistic behaviors and hybrid evolutions, i.e., stochastic hybrid system[7]. In order to verify the properties on such systems, e.g., reachability properties, one can translate the behaviours of the systems to an SSMT-C formula:

\[ Q : I \land T \land G \]

where \( Q \) is a sequence of existential and randomized quantifiers, \( I \) is the initial condition, \( T \) represents the transition relations and \( G \) is the goal which we hope the system can achieve. The properties can be quantitatively verified by compute the probability of satisfaction.

References


Trace Refinement of π-Calculus Processes∗

Manuel Gieseking (manuel.gieseking@informatik.uni-oldenburg.de)

Correct System Design, Carl von Ossietzky University of Oldenburg, Germany

Classically, the general focus while investigating the π-calculus [4] is put on bisimulations between processes to determine their behavioral equivalence [5]. From a somewhat more application-oriented point of view, it is of interest to analyze process algebras in terms of the so-called refinement relation. That is to determine whether an implementation satisfies its specification. This approach has for instance been investigated in the context of Tony Hoare’s CSP (Communication Sequential Processes [2]).

Both algebras are widely accepted for modeling and analyzing the communication between concurrent systems. Even though they have similar capabilities like parallel composition of processes, choice operators and actions/events which can be performed, they differ in some main features like the mobility aspect of the π-calculus to establish new communication by sending channels over channels.

Tony Hoare wrote in 2006 about CCS, the foundation of the π-calculus without the mobility aspect: “[…] the time has come to unify the two modelling styles [(CSP and CCS)], to enable practicing engineers to exploit a combination of their complementary advantages”1. Following this approach, we investigate in our work how the refinement notions of CSP can be applied to the π-calculus to exploit their advantages. Therefore, we present a new denotational semantics for the monadic π-calculus with guarded choice and without match and mismatch prefixes [1]. This approach is inspired by the trace semantics of CSP, as presented in [6] for example. Our further contributions are the proofs of compositionality results of the semantics, proofs establishing relations between trace refinement and (weak and strong) simulations, as well as algebraic laws and additional properties of the trace semantics for example concerning applications of substitutions and transpositions of names.

We choose the early transition system [7] as the foundation of our semantics, where every possible behavior a process can have is encoded in the label of the transitions. Thus, we can define a big-step semantics, similar to approaches for CSP, by collecting the labels of the transitions while following a path through the transition system. Those sequences of labels are also called traces. They hide the internal steps, that is, the internal communication of the process, since we would like to relate processes by their behavior visible to the environment. By collecting all possible traces of a process, we are obtaining the trace semantics. Instead of using the processes themselves we take their equivalence

*Acknowledgements: I would like to thank Sven Linker and Ernst-Rüdiger Olderog for being my supervisors and guides during the work of the underlying Master’s thesis. Especially Sven, thank you for your patience and the hours of fruitful discussions.

1[3], page 209.
classes (denoted by \([P]\) for a process \(P\)) with respect to the renaming of bound names within the process. The *bound names* are those occurring as an object of an *input prefix* (for example the name \(x\) in the process \(a(x).P\)) or as a restricted name within a process (that is, for instance, the name \(x\) in \(\mathtt{new} \ x \ P\)). For a process \(P\) evolving with a big-step to \(Q\) by using the trace \(t\), we write \([P] \xrightarrow{t} [Q]\). The trace semantics for a process \(P\) is denoted by \(\mathcal{T}([P])\) and a trace itself is written \(t = (\alpha_1, \ldots, \alpha_n)\), with only visible actions \(\alpha_i\). Thus, we can now define a condition (the specification) and check whether another process (the program) meets this specification by calculating the process’ semantics and checking set-inclusion. We say a process \(Q\) refines \(P\) (written \(P \sqsubseteq^\tau Q\)) if and only if \(\mathcal{T}([Q]) \subseteq \mathcal{T}([P])\).

As an example we consider an abstract communication protocol, where Alice \(A\) sends a message \(m_1\) to Bob \(B\) over a server \(S\) via a private channel \(c_1\).

\[
\begin{align*}
COM & = \text{def} A \mid S \mid B \\
A & = \text{def} \mathtt{new} \ c_1 \ (\overline{\tau}(c_1).\overline{\tau}(b).\overline{\tau}(m_1)) \\
S & = \text{def} s(c).c(r).c(m).(\overline{\log}(m).\tau(m) + \tau(m)) \\
B & = \text{def} b(m_2).\overline{\tau}(m_2)
\end{align*}
\]

First of all, Alice can establish a connection to the server via channel \(s\) by sending it the private channel \(c_1\), which is further used for the communication between Alice and the server. Then Alice sends the name of the recipient \(b\) and the message \(m_1\) over this private channel. Afterwards the server has the possibility to log the message and subsequently sending the received message to Bob or directly delivering it. Bob only listens at his address for some message and then does something not further specified with this message. This desired behavior is visualized through an extract of the process’ operational semantics within Fig. 1. The internal communications are denoted by \(\tau\)-transitions. The transitions which are used for calculating the visualized big-step with its trace \(\tau(m_1), \overline{\tau}(m_1)\), are the transitions of the left branch of the given extract of the transition system.

The trace semantics does not only consider the behavior resulting from the synchronized communication of the processes, but also collects all interleavings of the visible actions the constituent processes can perform. Figure 2 visualizes an extract of Alice’s actions. Remark that the transitions labeled with \(\overline{\tau}(x)\), where \(x\) may be any name, which does not occur free in the communication process, exist by reason of considering equivalence classes of the processes.

By collecting all of those traces – meaning all paths and all of their prefixes – in one set, we are gaining the trace semantics. That is for a process \(P\) the trace semantics is defined by

\[
\mathcal{T}([P]) = \left\{ t \in \text{Traces} \mid \exists Q \in \mathcal{P}^\tau : [P] \xrightarrow{t} [Q] \right\},
\]

where \(\mathcal{P}^\tau\) is the set of all processes and \(\text{Traces}\) the set of all traces. If we additionally consider a server without logging \((S' = \text{def} s(c).c(r).c(m).\overline{\tau}(m))\), we can prove by set-inclusion that the protocol without logging refines the other, since it has less behavior than the logging one.

The compositionality of the CSP trace semantics is one advantage which we tried to preserve by the definition of our denotational semantics. That is, for each syntactical
operator of the $\pi$-calculus we would like to have a semantical counterpart, such that the semantics of the composition of the syntactical processes is equal to the composition of the semantics of the syntactically composed parts. For instance, we try to find for a binary syntactical operator $\circ$ a semantical one $\circ_T$ with $T([P \circ Q]) = T([P]) \circ_T T([Q])$. We show that the $\tau$ and output prefix, as well as the choice operator are compositional similar to their corresponding operators in CSP. Thus, we can easily show that for given processes $P, Q \in \mathcal{P}^\pi$ and names $a, x \in \mathcal{N}$

$$ T([\tau.P]) = T([P]) \quad \text{(TAU)} $$

$$ T([\overline{a}(x).P]) = \{\} \cup \{\langle \overline{a}(x) \rangle \} \sim T([P]) \quad \text{(OUTPUT)} $$

$$ T([P + Q]) = T([P]) \cup T([Q]) \quad \text{(CHOICE)} $$

hold. Where $\sim$ is the concatenation of the traces within the given sets, $\langle \rangle$ the empty trace without containing any visible behavior, and $\mathcal{N}$ the set of all possible names/channels. But due to the specialty of the $\pi$-calculus permitting new communication to be established by transmitting a channel via an input process, the compositionality of the input prefix is limited. Given a process $P \in \mathcal{P}^\pi$ and names $a, x \in \mathcal{N}$, then

$$ T([a(x).P]) = \{\} \cup \{\langle a.y \rangle \sim s \mid s \in T([P\{y/s\}]), y \in \mathcal{N}\} $$

Figure 1: An extract of the early transition system of the communication between Alice and the server.
Figure 2: An extract of the early transition system of Alice’s sole behavior.

holds. Where \( \{y/x\} \) is the substitution of the name \( x \) with the name \( y \) in the process \( P \). Since the application of the substitution is within the trace semantics, we have not obtained a compositional definition of the behavior of an input process. However, at least for every fresh name received by an input process the calculation of the traces is compositional. This stems from the equality of a transposition and a substitution of free names and the equality of the trace semantics by applying a transposition of names to the process within the semantics or to the semantics itself. That is, for a process \( P \) and a transposition \( \sigma \) the equality \( T([P\sigma]) = T([P])\sigma \) holds. Furthermore, we show that the parallel composition is compositional for processes without any occurrence of restriction operators inside. The proof exploit a Gödel numbering of the traces and a new inductively defined trace calculation of the parallel composition [1]. To show that the limitation to restriction free processes for the compositionality of the parallel composition is still feasible, we extend Milner’s structural congruence, such that every restriction operator can be moved right at the front of the process. That is, we add rules still preserving the congruence properties to extend the scope of a restriction over prefix and choice operators. We show that the limitation to restriction free processes within the compositionality of the parallel composition is not harmful, since we offer an extended standard form, to which every process can be transformed such that the resulting process is still structural congruent to the original one. A process in extended standard form has all restriction operators right at the front of the process. The proof of transforming a process in extended standard form is constructive and since it does not increase or reduce the number of operators, the transformation does not incur a large blowup. Furthermore, we show that the restriction operator is most likely compositional, but the proof of the one set-inclusion is not yet completely finished.

Additionally, we sketch an algorithmic idea for calculating the trace semantics for a finite set of names provided by the environment. This approach takes advantage of the
compositional aspects of the semantics.

Moreover, we show that the trace semantics preserves some algebraic laws and useful properties. For instance, we show that the prefix operator distributes over the choice operator, that is, $T([\pi.M_1 + \pi.M_2]) = T([\pi.(M_1 + M_2)])$, and state that $M_1 \sqsubseteq_T M_2$ and $M_3 \sqsubseteq_T M_4$ implies $M_1 + M_3 \sqsubseteq_T M_2 + M_4$ for an arbitrary prefix $\pi$ and sums $M_1, \ldots, M_4$. Furthermore, we prove that $Q \sqsubseteq_T P$ is equivalent to $\overline{\pi}(x).Q \sqsubseteq_T \overline{\pi}(x).P$ for all processes $P, Q$ and names $a, x$.

Finally, we connect our new approach to the well-known concepts of weak and strong simulations. Thus, we prove that a weak as well as a strong simulation imply the trace refinement of every element of the simulation. That is, if a process $Q$ simulates $P$ (strongly or weakly, respectively), then the process $P$ also refines $Q$. Furthermore, the strong or weak bisimilarity of processes implies their trace equivalence.

References


We consider sequential probabilistic programs, which are a means to model randomised algorithms in computer science. These algorithms occur in many areas within computer science such as machine learning [3], artificial intelligence [11], security [2] or randomised algorithms design [9]. A probabilistic programming language allows for the straightforward specification of algorithms and moreover, it facilitates the formal analysis of algorithms. For instance, deductive verification techniques, which rely on invariants and may be mechanised using theorem proves, can be applied to establish the correctness of probabilistic programs. Here we choose to work with the probabilistic Guarded Command Language (pGCL), which is a probabilistic extension of Dijkstra’s GCL [5], and was introduced by McIver and Morgan [8]. A program written in this language can draw a sample from a Bernoulli distribution and, depending on the outcome, executes one or the other branch of a choice statement. A common illustration of a Bernoulli experiment is a coin flip which has two outcomes “heads” or “tails”. Of course, a Bernoulli experiment need not have equal probabilities for both outcomes and then in our illustration we speak of a biased coin flip. Having only Bernoulli trials at our disposal might seem to be a severe limitation, but in fact this is sufficient to write subprograms that produce a sample from e.g. geometric, binomial or uniform distributions.

In probability theory, it is common to condition the probability of an event or the expectation of a random variable on the occurrence of some other event. In this way one obtains conditional probability distributions and conditional expectations. An application of conditional probabilities can, for instance, be found in medicine where one tries to estimate the likelihood of a particular disease after having observed some symptoms. In a similar way, we may ask for the expected outcome of a probabilistic program given the fact that it has visited particular states during its execution. To allow for such specifications we follow Claret et al. [4] and add the observe keyword to the pGCL language. Consider the following program for example.

*This abstract is based on a paper, currently under review, that has been jointly written with Nils Jansen, Benjamin Kaminski, Joost-Pieter Katoen, Annabelle McIver and Federico Olmedo.*
This program models an example taken from [12, p. 216]. A fishbowl contains two fishes \( f_1 \) and \( f_2 \) where we only know that \( f_1 \) could equally likely be a piranha or a goldfish while \( f_2 \) is surely a piranha. Randomly one of the fish is removed from the fishbowl and observed to be a piranha. We can ask for the probability that \( f_1 \) is a piranha given our observation that the sample fish is a piranha. This happens to be the case with probability \( \frac{2}{3} \). In [4] and related work, e.g. [7, 10], they are concerned with purely probabilistic programs for which they try to find the probability of some outcome using simulation or symbolic program execution. All their programs are assumed to be terminating almost surely. Our contribution is to

1. provide semantics without making assumptions about the termination behaviour of the program and link operational and denotational semantics for fully probabilistic programs,
2. explain why denotational semantics cannot be defined inductively for non-deterministic programs,
3. and finally, we introduce program transformation that allow to eliminate conditioning

For the lack of space we exemplify the aforementioned points while skipping all technical details.

**Semantics** Earlier work [6] shows how probabilistic programs may be understood as a (possibly infinite) reward Markov decision process (RMDP). For a program \( P \), the RMDP \( \mathcal{R}_f[P] \) models the stepwise behaviour of \( P \) and assigns rewards to the final states according to a given reward function \( f \). Thus the average outcome of the program \( P \) with respect to \( f \) is determined by the minimal expected reward in \( \mathcal{R}_f[P] \).

Now we can generalise this approach to conditional expectations. We only need to add one rule to the structured operational semantics (SOS) in [6] in order to construct an RMDP for a program with \texttt{observe} statements. This new rule translates \texttt{observe} statements as a deterministic step without any effects (just like \texttt{skip}), however if the observations does not hold in the current state, this state is labelled with a \( \Diamond \). Then, the average outcome of \( P \) with respect to \( f \), given that all observations are passed can be easily calculated as follows: as before, we determine the expected reward in \( \mathcal{R}_f[\ ] \), but now we divide it by the probability of all runs that never violate an observations. Due to the presence of non-deterministic choices this quotient has to be minimised across all schedulers. Formally the conditional minimal expected reward until reaching \( T \) from \( s \) avoiding \( \Diamond \) states, denoted \( \text{CExpRew}^{(M,P)}(s \models \Diamond T \models \Diamond \Diamond) \), is defined by:

\[
\inf_{\mathcal{S}} \sum_{c \in C} c \cdot \Pr_{\mathcal{S}} \{ \pi \in \text{Paths}^\mathcal{S}(s, \Diamond T) \mid r_T(\pi) = c \} \cdot \frac{1 - \Pr_{\mathcal{S}} \{ \pi \in \text{Paths}^\mathcal{S}(s, \Diamond \Diamond) \}}{1 - \Pr_{\mathcal{S}} \{ \pi \in \text{Paths}^\mathcal{S}(s, \Diamond \Diamond) \}}.
\]
An alternative way to determine expected outcomes of a program $P$ is to consider its denotational semantics in terms of expectation transformers. For this McIver and Morgan [8] introduced the notions of the greatest pre-expectation $wp(P, \cdot)$ and greatest liberal pre-expectation $wlp(P, \cdot)$ which are quantitative generalisation of Dijkstra’s predicate transformers [5]. In earlier work [6] we have established a correspondence between the minimal expected reward in $Rf[\{P\}]$ and $wp(P, f)$. Recently we were able to generalise this result to conditional expectations and have shown that the conditional minimal expected reward of program $P$ and reward function $f$ amounts to $cw_p(P, f) = wp(P, f) = wlp(P, 1)$ for fully probabilistic programs $P$. However if the program text in $P$ contains non-deterministic choices it becomes impossible to determine the conditional expectation using $wp$ or $wlp$.

This is the result of the next section.

No inductive definition of for non-deterministic programs In the following we illustrate why positional schedulers are not sufficient to determine the conditional expected reward in an RMDP. This is closely related to a result due to [1]. As a consequence we do not have a $cw_p$ rule for non-deterministic choice that tells us how to compute the minimal conditional expectation from the current valuation and the $cw_p(P, \cdot)$ and $cw_p(Q, \cdot)$ of the two subprograms $P$ and $Q$.

Consider the RMDP $R$ in Figure 1. There are only two schedulers. Let $\mathcal{S}_\mu$ be the scheduler that chooses to go from $s_2$ to $s_3$ and let $\mathcal{S}_\nu$ be the scheduler that chooses $s_4$ as the successor of $s_2$. Further let $T = \{s_1, s_3, s_6\}$ and let $R_{\mathcal{S}}$ be the Markov chain obtained from $R$ by resolving all choices according to $\mathcal{S}$. Then we calculate

$$CExpRew^{R_{\mathcal{S}_\mu}}(\diamond T | \neg \diamond \mu) = 1.5 \text{ and } CExpRew^{R_{\mathcal{S}_\nu}}(\diamond T | \neg \diamond \nu) = 1.4.$$

Hence $CExpRew^R(\diamond T | \neg \diamond U) = 1.4$ and the minimising scheduler is $\mathcal{S}_\nu$. However if we only consider the subsystem $R'$ that starts execution in state $s_2$ we obtain

$$CExpRew^{R_{\mathcal{S}_\mu}}(\diamond T | \neg \diamond \mu) = 2 \text{ and } CExpRew^{R_{\mathcal{S}_\nu}}(\diamond T | \neg \diamond \nu) = 2.2.$$

So in that subsystem the minimising choice is given by $\mathcal{S}_\mu$. This shows how choices are resolved depending on the “context” within which the state occurs in the system. As mentioned above, Andrés et al. [1] show that a particular class of history dependent schedulers is needed to find the conditional expected reward.

This example shows that any attempt to define any transformer $T$ for non-deterministic choice as

$$T(\{P\} \Box \{Q\}, X) = \min_{<} \{T(P, X), T(Q, X)\}$$
must fail for any representation of conditional expectations \(X\) and any order \(\preceq\) between them because the decision is made at a point where the “context” information is missing. In this sense no inductive definition of a conditional expectation transformer is possible as was the case for \(wp\) and \(wlp\).

**Transformations**  Probabilistic programs with conditioning can be analysed using the operational or denotational semantics as described above. However in some instances we may simplify the analysis by removing the observations entirely and determining the unconditioned expected reward on the transformed program. For this we propose two approaches. One is based on the insight that observations can be “pushed” upwards in the program text. The idea is that it is possible to pre-compute the probability that all observations are passed starting in a given initial state. This precomputation is carried out regardless of the reward function at hand. Thus the conditional expected reward is then simply determined as the (unconditioned) expected reward on the transformed program.

The second proposed transformation restarts a program run every time it violates some observation. Thus only those runs terminate and contribute to the outcome of the program that pass all observations on their way. This transformation can be carried out purely syntactically but may complicate the analysis as it introduces an extra loop.

**References**


Markov Reward Automata in Railway Engineering

Dennis Guck (d.guck@utwente.nl)

Formal Methods and Tools, Faculty of EEMCS
University of Twente, The Netherlands

1 Introduction

RAMS (Reliability, Availability, Maintenance, Safety) requirements are utmost important for safety-critical systems like the railway infrastructure. Furthermore, costs and rewards are important ingredients in the analysis of many of those systems, e.g. modelling critical aspects like energy consumption, task completion, repair costs, and memory usage. For example in railway engineering typical optimisation questions are: Should we carry out preventive maintenance to prevent future costs? Can we reduce the down-time during a maintenance procedure? How can we schedule the maintenance task such that the operational costs are minimised? Such optimisation questions incorporate typically the following ingredients: (1) stochastic timing to model degeneration or delay; (2) discrete probability to model random failures; (3) nondeterministic choices to model uncertain behaviour; (4) costs/rewards to measure the quality of a solution.

Markov automata (MAs) have been introduced in [6] as a continuous-time version of Segalas probabilistic automata [11]. They provide a formalism for modelling systems incorporating continuous stochastic timing, discrete probabilistic choices as well as nondeterministic choices. They provide a well-defined semantics for generalised stochastic Petri nets (GSPNs) [5], dynamic fault trees [2] and the domain-specific language AADL [3]. Moreover, recent work demonstrated that MAs are suitable for modelling and analysing distributed algorithms such as a leader election protocol, performance models such as a polling system and even hardware models such as a processor grid [12]. Furthermore, in [9] the model was extended with rewards to Markov reward automata (MRAs), such that it is now possible to incorporate instantaneous transition rewards as well as time-based state rewards.

2 Approach

In recent work we investigated quantitative analysis of MAs [7] as well as their extension to rewards [9]. We extended the formalism of Markov reward automata with two reward functions: a transition-reward function and a state-reward function. The transition-reward

*This work has been supported by the STW-ProRail partnership program ExploRail under the project ArRangeer (12238), and we acknowledge our cooperation with Movares in the ArRangeer project.
function assigns a rational number to each transition, representing an instantaneous reward that is obtained directly when taking that transition. The state-reward function assigns a rational number to each state, representing a reward that is obtained over time during a stay of one time unit in that state.

**Definition 2.1 (Markov reward automata).** A *Markov reward automaton (MRA)* is a tuple $\mathcal{M} = (S, s^0, A, T, \rho)$, where

- $S$ is a countable set of *states*, of which $s^0 \in S$ is the *initial state*;
- $A \subseteq \text{Act}$ is a countable set of *actions*;
- $T \subseteq S \times A \times \mathbb{R}_{\geq 0} \times \text{Distr}(S)$ is the *transition relation* including *transition rewards*;
- $\rho: S \to \mathbb{R}_{\geq 0}$ is the *state-reward function*.

We require for each $s \in S$ that there is at most one transition labelled with $\chi(\cdot)$\(^1\). Further, we require that $T$ is countable and write $s \xrightarrow{\alpha} r, \mu$ if $(s, \alpha, r, \mu) \in T$.

The function $\rho$ associates a real number to each state. This number may be zero, indicating the absence of a reward. The state-based rewards are gained *while being in a state*, and are proportional to the duration of this stay. The transition-based rewards are gained instantaneously *when taking a transition* and are included directly in the transition relation.

Rewards can be used to model many quantitative aspects of systems, like energy consumption, memory usage, deployment or maintenance costs, etc. The total reward of a path (e.g., total amount of energy consumed) is then obtained by adding all rewards along that path, that is, all state rewards multiplied by the sojourn times of the corresponding states plus all transition rewards on the path.

In [7] we presented a framework for modelling and quantitative analysis of timed and long-run objectives of MAs. Following this work, we lifted the existing framework to the realm of rewards [9]. There we introduced MRAs and extended the algorithms for computing time-bounded, goal-bounded and long-run objectives to rewards. Further, the MAMA tool-chain consisting of SCOOP and IMCA was extended to incorporate all changes.

### 3 Quantitative analysis

The three main aspects of the quantitative analysis of MRAs that we consider are: (1) the expected time/reward to reach a set of goal states, (2) the expected probability/reward until a given time bound, and (3) the long-run average time/reward of a set of states. Typical examples where those objectives are important are respectively: to minimise the average energy consumption of a server farm; to minimise the average maintenance cost of a railway line over the first year of deployment; and to maximise the yearly revenues of a data center over a long time horizon. In the following subsections we give a short summary of the results of the quantitative analysis form [7] and [9].

\(^1\)The actions $\chi(\cdot)$ represent exit rates and are used to distinguish probabilistic and Markovian transitions.
3.1 Goal-bounded objectives

In the goal-bounded reachability we are interested in the minimal and maximal expected cumulative reward gained until reaching a set of goal states $G \subseteq S$. Therefore, we accumulate the state and transition rewards until a state in $G$ is reached; if no state in $G$ is reached, we keep on accumulating rewards. The expected cumulative reward can be formalized into a classical Bellman equation. A direct consequence of this is, that the expected cumulative reward is attained by a stationary deterministic policy. In case we are interested in the expected time, we have to assign each transition a reward of 0 and each state a reward of 1.

3.2 Time-bounded objectives

The time-bounded objectives can be distinguished in: (1) probability of reaching a set of goal states in time $T$; (2) accumulated reward until time $T$. Both problems can be generalised as a fixed point characterisation (FPC) similar to [14]. However, the FPC is not algorithmically tractable and needs to be discretised. In both cases we will discretise the computation in the following manner. We split the time interval into equally-sized discretisation steps, each of length $\delta$. Thus, we divide the time horizon $[0, b]$ into a generally large number of equidistant time steps, each of length $0 < \delta < b$, such that $b = k\delta$ for some $k \in \mathbb{N}$. The discretisation step is assumed to be small enough such that with high probability it carries at most one Markovian transition. The advantage of this discretisation is, that we can specify the desired error bound a priori and based on that adjust the discretisation step. A time-dependent deterministic policy is sufficient for the time-bounded reachability. That is, the policy decides on the basis of the states visited so far and their timing.

3.3 Long-run objectives

In the long-run average objectives we are interested in the average cumulative reward induced by a set of goal states $G \subseteq S$ in the long-run as well as long-run average fraction of time spent in $G$. The computation can be split up in a three step procedure:

1. Determine the maximal end components\(^2\) \{$M_1, \ldots, M_k$\} of MRA $M$.
2. Determine the long-run average in maximal end component $M_j$ for all $j \in \{1, \ldots, k\}$.
3. Reduce the computation of the long-run average in MRA $M$ to a stochastic shortest path (SSP) problem.

The first phase can be performed by a graph-based algorithm [4], whereas the last two boil down to solving distinct LP problems. For determining the long-run average in each MEC, we can reduce the problem to the long-run ratio objective for Markov decision processes [1]. The last phase is then a SSP computation, where the long-run averages from phase

\(^2\)A sub-MRA $M$ is a pair $(S', K)$ where $S' \in S$ and $K$ is a function that assigns to each state $s \in S'$ a non-empty set of actions, such that for all $a \in K(s)$, $s \xrightarrow{a} \mu$ with $\mu(s') > 0$ implies $s' \in S'$. An end component is a sub-MRA whose underlying graph is strongly connected; it is maximal (a MEC) w.r.t. $K$ if it is not contained in any other end component $(S'', K)$. 

123
two are used as goal costs. As a consequence of this, a stationary deterministic policy is sufficient to compute the long-run average.

4 Reliability engineering

Fault tree analysis (FTA) [13] is often used in industry as part of their RAMS analysis. Fault trees (FTs) model the failure propagation throughout a system: the leaves are basic events (BEs) and represent component failures and the other nodes express the failure propagation via AND and OR gates. Shortly summarized, FTA yields measures for system availability and reliability, however it lacks the capability to incorporate maintenance procedures. In [8] we presented a first step of incorporating simple maintenance analysis to the FTA approach from [2]. Note that we consider dynamic fault trees (DFTs), they extend standard FTs with a number of intuitive gates and facilitate the modelling of often recurring concepts in reliability engineering, i.e. spare management, functional dependencies, and order-dependent behaviour. Furthermore, the underlying quantitative model used for analysis are input-output interactive Markov chains [10].

Extending the framework to MRAs will give us several advantages. On the one hand we can incorporate down-time, repair, and maintenance costs into the model. Thus, we can use those metrics to find the optimal and most cost efficient maintenance strategy. On the other hand we can combine continuous and probabilistic behaviour of a system by keeping the modularity of the FTA approach from [2]. Thus, new events like a probabilistic installation error can be introduced to the FT. This will give the engineers the possibility for a more accurate description of the fault behaviour of a system.

5 Conclusion and Challenges

We presented Markov reward automata (MRAs), an extension of Markov automata (MAs) featuring both transition-based and state-based rewards. This formalism allows to model a wide variety of systems featuring nondeterminism, discrete probabilistic choice, continuous stochastic timing and transition-based and state-based rewards. Furthermore, we presented several algorithms for quantitative analysis of MRAs, including expected reachability properties as well as long-run average properties.

The integration of a more realistic FT behaviour including maintenance strategies using MRAs is currently in development. Furthermore, future work on the quantitative analysis of MRAs will be on negative rewards, more complex optimisation criteria, as well as the handling of several rewards as multi-optimisation problem are important topics for future research.

References

Markov Reward Automata in Railway Engineering


Weighted Unranked Tree Automata
over Tree Valuation Monoids*

Doreen Heusel (dheusel@informatik.uni-leipzig.de)

Institut für Informatik, Universität Leipzig, D-04109 Leipzig, Germany

Recently, the investigation of formal languages of unranked trees has found much attention due to the development of the modern document language XML and the fact that (fully structured) XML-documents can be formalized as unranked trees. With the help of unranked tree automata, one can investigate qualitative questions on XML-documents. To allow the study of quantitative aspects, Droste and Vogler [8] introduced and investigated weighted automata on unranked trees over semirings. These weighted automata enable us, for instance, to evaluate the consumption of resources in systems, to assess the reliability of transitions, or to model real-valued degrees of truth values of events.

Weighted logics over semirings represent another common approach for the investigation of quantitative aspects. For words, a weighted MSO logic which is expressively equivalent to weighted word automata was already developed in 2005 by Droste and Gastin [3]. Analogous formalisms followed for various structures like for infinite words, for ranked trees, for infinite trees, for trace languages, for picture languages, for texts and for nested words.

In order to obtain a logic counterpart for their weighted unranked tree automata, Droste and Vogler [8] presented a weighted MSO logic for unranked trees. Their automata capture the expressiveness of their logic but surprisingly not the other way around. Droste and Vogler stated as an open problem to determine a weighted automata model expressively equivalent to their logic. Our goal is to attack this problem.

For this, we present a new class of weighted tree automata. Syntactically they do not differ much from the ones of [8]. They still consist of a state set and a family of weighted word automata which are used to calculate the local weight of a position of an input tree by letting the weighted word automaton associated with this position run on the labels of the children of the position. The conceptual differences are the following: for the behavior definition of the weighted unranked tree automata over valuation monoids, we do not use runs anymore. Instead we fall back on the technically more involved extended runs, which were already introduced in [8]. Besides from the information of classical runs, extended runs also collect a run per weighted word automaton called for every position of the input tree. In addition we change the way how the weight of such an extended run is calculated. For weighted unranked tree automata over semirings, the local weight of a position is defined by the weight of the run chosen for the word emerged of its children’s labels. Here the local weight of a position equals the weight of the transition taken for this position in the run of the position’s parent.

*Partially supported by DFG Graduiertenkolleg 1763 (QuantLA).
Moreover we consider tree valuation monoids as weight structures which resort to the ideas of Chatterjee, Doyen, and Henzinger [2] that were already generalized to valuation monoids for words by Droste and Meinecke [7]. Tree valuation monoids are additive monoids equipped with a valuation function that assigns any tree whose labels are stemming from the additive monoid to a value from this monoid. We will use the valuation function to calculate the weights of a run in a global way, i.e. given a run we apply the valuation function to all local weights which appear along the run. Tree valuation monoids are a very general structure. They contain all semirings, all bounded (possibly non-distributive) lattices, which occur in multi-valued logics, and in addition they enable us to cope with non-binary evaluation functions like average or discounting.

Now our main results are the following.

• With the new definition of weighted unranked tree automata over tree valuation monoids based on extended run we obtain a model that subsumes the weighted unranked tree automata over commutative semirings as well as the weighted ranked tree automata over tree valuation monoids we introduced in [5].

• We show that our weighted unranked tree automata are closed under the common operations like sum, product and relabeling.

• We define a weighted MSO logic for unranked trees over tree valuation monoids analogously to the one for words over valuation monoids from Droste and Meinecke [7] and characterize the behavior of our weighted unranked tree automata by four different fragments of the logic. Which fragment can be used depends on purely syntactically restrictions on the underlying tree valuation monoid. Thereby we solve the open equivalence problem of Droste and Vogler.

• We investigate weighted unranked tree automata over a special class of tree valuation monoids; the locally finite tree valuation monoids. Here a restriction of the weighted MSO logic is unnecessary in order to obtain an equivalence result to the weighted unranked tree automata if some additional assumptions on underlying valuation monoids are fulfilled.

Our penultimate main result indicated above generalizes both the respective results of [8] about weighted unranked tree automata over commutative semirings and the results we got about weighted ranked tree automata over tree valuation monoids (cf. [5]).

At the end, we investigate the supports of weighted unranked tree automata (to be published in [6]). The support of a weighted automaton is defined as the language of the underlying structures (strings, trees, ...) which the weighted automaton evaluates to non-zero. We show that the support of a weighted unranked tree automaton over a zero-sum free, commutative strong bimonoid is recognizable. For this, we use methods of Kirsten [9], in particular, his construction of finite automata recognizing the supports of weighted automata on strings over zero-sum free, commutative semirings. We also get an effective construction of a finite tree automaton recognizing the support of a given weighted unranked tree automaton for zero-sum free, commutative strong bimonoids where Kirsten’s zero generation problem is decidable. In addition, we give a translation of nested weighted automata into weighted unranked tree automata for arbitrary commutative strong bimonoids. As a consequence, we derive the main result of [4] on the support of nested
weighted automata and the present support result. Our result on the support of nested weighted automata together with the results of Bollig, Gastin, Monmege, and Zeitoun [1] partially solve the open problem “to determine for which semirings the satisfiability problem is decidable” which was stated by Bollig et al. in [1].

References


1 Negotiations

In [7, 8], Javier Esparza and Jörg Desel have introduced a model of concurrency with multi party negotiation as primitive. The model allows one to describe distributed negotiations obtained by combining “atomic” multi party negotiations, or atoms. Each atom has a number of parties (the subset of agents involved), and a set of possible outcomes. The parties agree on an outcome, which determines for each party the subset of atoms it is ready to engage in next.

1.1 Soundness

Ill-designed negotiations may deadlock, or may contain useless atoms, i.e., atoms that can never be executed. The problem whether a negotiation is well designed or sound was studied in [7, 8]. The main result was that for a class called deterministic negotiations, there is a reduction procedure that decides the soundness problem in polynomial time and for a class called acyclic weakly deterministic negotiations a similar procedure can be applied which is conjectured to also be polynomial.

In my thesis, we study the approach of [7] which involves summarizing a negotiation, that is, reducing the number of atoms to one while maintaining the possible transformations of the agent’s internal states. It is shown that a negotiation can be summarized with the given reduction procedure if and only if it is sound. We analyze their proofs and results and point out oversights. For each such flaw, we provide a correction in form of modified definitions, rephrased theorems or new proofs.

It turns out that our proposed changes to [7] not only correct errors but also allow us to extend their results: The extended rules we provide can also summarize acyclic weakly deterministic type 2 negotiations, a new, wider class of negotiations that we will introduce. Furthermore, we relax the concept of cyclic negotiations and are able to generalize the result to weakly deterministic negotiations in which only the deterministic agents are acyclic.

We also give a summarization procedure that is able to summarize all acyclic negotiations. This procedure is based on a different approach: Instead of applying the reduction
rules until a summary is reached, there is also a final step that is not a rule but a “clean up” after the rules have processed the negotiation.

The provided reduction algorithms of [7, 8] that we also study avoid the construction of the state space. Instead, syntactic reduction rules are applied exhaustively to simplify the system step by step while simultaneously maintaining important aspects of the behavior like absence of deadlocks. For Petri nets or workflow nets this approach has been studied for quite some time, mostly for the liveness and soundness problems [2, 12, 13]. For these problems there are various reduction rules known, and they have been proven complete for certain classes of systems [11, 5, 6], meaning that they reduce all live or sound systems in the class, and only those, to a trivial system (in our case to a single atomic negotiation). However, for the summary problem many of these rules, like the linear dependency rule of [6], are unfit as they do not preserve all behaviors, but only the soundness property.

The rules presented in [7] which we extend and modify form a complete set of reduction rules for the summarization problem of acyclic negotiations that are either deterministic or weakly deterministic. These rules are inspired by reduction rules used to transform finite automata into regular expressions by eliminating states [14].

1.2 Games

We furthermore start the study of games on negotiations. As for games played on pushdown automata [17], vector addition systems with states (VASS) [3], counter machines [15], or asynchronous automata [16], games on negotiations can be translated into games played on the (reachable part of the) state space. However, the number of states of a negotiation may grow exponentially in the number of agents, and so the state space can be exponentially larger than the negotiation. We explore the complexity of solving games in the size of the negotiation, not in the size of the state space. In particular, we are interested in finding negotiation classes for which the winner can be decided in polynomial time, thus solving the state space explosion problem.

We study games formalizing two interesting questions related to a negotiation. First, can a given coalition (i.e., a given subset of agents) force termination of the negotiation? (Negotiations may contain cycles.) Second, can the coalition force a given final outcome?

Our first results show that these two problems are EXPTIME-complete in the size of the negotiation. This is the case even if the negotiation is deterministic, and so it seems as if the tractability results of [7, 8] cannot be extended to games. However, we are able to show that, surprisingly, the problems are polynomial for deterministic (or even weakly deterministic) negotiations that are sound. This is very satisfactory: since unsound negotiations are ill-designed, they are not of interest to us anyway. And, very unexpectedly, the restriction to sound negotiations has as collateral effect a dramatic improvement in the complexity of the problem. Moreover, the restriction comes “at no cost”, because deciding soundness of deterministic negotiations is also decidable in polynomial time.

Our games can be seen as special cases of concurrent games [1, 4] in which the arena is succinctly represented as a negotiation. Explicit construction of the arena and application of the algorithms of [1, 4] yields an exponential algorithm, while we provide a polynomial one.

Negotiations have the same expressive power as 1-safe Petri nets or 1-safe VASS,
although they can be exponentially more compact (see [7, 8]). Games for unrestricted
VASS have been studied in [3]. However, in [3] the emphasis is on VASS with an infinite
state space, while we concentrate on the 1-safe case.

The papers closer related to ours are those studying games on asynchronous automata
(see e.g. [16, 9, 10]). Like negotiations, asynchronous automata are a model of distributed
computation with a finite state space. These papers study algorithms for deciding the
existence of distributed strategies for a game, i.e., local strategies for each agent based
only on the information the agent has on the global system. Our results identify a special
case with much lower complexity than the general one, in which local strategies are even
memoryless.

In my thesis, we study earlier work [7] and analyze the approaches and results given.
We therefore repeat definitions, theorems and proofs for readability and in-depth analysis.
The games chapter is an extension of an arXiv paper with the title “Negotiation Games”
by the author of this thesis and Javier Esparza.

References


and G. Rozenberg, editors, Advances in Petri Nets, volume 254 of LNCS, pages

systems with states. In S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der
Heide, and P. G. Spirakis, editors, ICALP (2), volume 6199 of Lecture Notes in


York, NY, USA, 1995.

and H. C. Melgratti, editors, CONCUR, volume 8052 of Lecture Notes in Computer

cyclic negotiations. In A. Muscholl, editor, FoSSaCS, volume 8412 of Lecture Notes


1 Introduction

In this paper I present Isabelle’s probability theory. The development is now at a point where we can formalize probabilistic semantics [8] (by employing the Giry monad), start to prove stochastic results (e.g. the Central Limit Theorem [2]), or analyse Markov decision processes [10, 21, 13, 12]. When developing these applications, we formalized a lot of probability theory itself. We generalized these new formalizations and moved them into our main probability theory. Usually, this happens in lockstep, i.e. new applications often force us to formalize new concepts or at least to generalize existing concepts. But then, new concepts allow us to tackle new applications.

2 Implementing Probability Theory

Measurability Many theorems in measure and probability theory assume the measurability of the occurring sets and functions. While for some measurable spaces (e.g. Borel sets) it is hard to construct a non-measurable sets or functions, measurability is still a side-condition we need to prove for each measure-theoretic rule we apply. I implemented a measurability prover for Isabelle: the user annotates measurability rules which are then used by the rewrite engine to prove measurability side-conditions. Measurability is compositional, so proving measurability is mostly concerned in finding the necessary rules.

For some constants, the measurability rule can get complicated. For example, measurability of the Lebesgue integral requires the measurable space of subprobabilities $\mathcal{S}(N)$ [9, 13], and the product space $M \times N$. In the following, $\mathcal{B}$ is the Borel space where all real intervals are measurable sets. To state that a function $f$ is measurable from $M$ into $N$ we write $f \in \mathcal{M}(M, N)$. With the space $\mathcal{S}(N)$ we express measurability of functions into (subprobability) measures, i.e. $(\lambda \mu. \mu A) \in \mathcal{M}(\mathcal{S}(N), \mathcal{B})$, for all $N$-measurable sets $A$. We use the product space $M \times N$ to express the measurability of functions depending on multiple variables. This can be used to give a measurability rule for the Lebesgue integral in its full generality:

$$
\frac{(\lambda(x, y). f x y) \in \mathcal{M}(M \times N, \mathcal{B}) \quad \mu \in \mathcal{M}(M, \mathcal{S}(N))}{(\lambda x. \int_y f x y \, d(\mu x)) \in \mathcal{M}(M, \mathcal{B})}
$$

*Supported by DFG project Ni 491/15-1
In most mathematical text books the measurability of integration is not provided in this
generality. Usually, the measure $\mu$ is not allowed depend on the variable $x$.

Besides the measurability rules and prover support, we also provide means to prove
measurability for (co)inductive predicates and (co)recursive functions. This is done by
reducing the definition to its least (resp. greatest) fixed point construction and proving
the continuity of the defining rules.

**Constructing Measures** We provide various means to construct measures. We for-
malized the classical constructions from measure theory, like products, counting space,
the Lebesgue measure, the push-forward measure of a measurable function, measures with
a density, or the limit of measures by invoking the Danielle-Kolmogorov extension theo-
rem [16]. These measures already provide a convenient way to construct new ones, e.g.
the normal distribution can be easily constructed by applying the bell curve as a density
to the Lebesgue measure.

A more modern construction we provide is the Giry monad [9], which provides a very
flexible way to compose measures. The bind-operator of the Giry monad can be seen as
integration over measures: $(\mu \gg \nu) A = \int \nu \cdot A \, d\mu$ for all $\nu \in \mathcal{M}(M, \mathcal{S}(N))$ and $A \in N$.
This allows us to define the semantics of probabilistic programming languages where the
choice of a random value depends on other random values.

**Bochner Integration** Lebesgue integration comes in two flavours: (1) as integration
over real-valued non-negative measurable functions, which is always defined, but the re-
sult can be infinity, and (2) as integration over integrable functions which are measurable
and where the positive and the negative part have a finite non-negative integral. The
second flavour can be generalized to Bochner integration: integration of functions into
Banach spaces (with a second-countable topology). We chose to formalize this general-
ization, which was well-supported by the availability of Banach spaces and the necessary

**Probability Mass Functions** While the previous results are all generalized for mea-
sures on arbitrary measurable spaces, an important subclass are the discrete measures.
They are described by probability mass functions (pmf), i.e. non-negative real-valued
functions which sum up to 1. From each pmf we can construct a measure and hence re-
use all results we formalized in measure theory. Andreas Lochbihler recently formalized
that the relator on pmfs preserves weak-pullbacks (i.e. the relator preserves transitivity:
[17]). This integrates the pmfs with Isabelle’s new datatype package [5].

3 Applications

**Central Limit Theorem** The central limit theorem is the only theorem in Isabelle
concerning *classic* stochastical analysis [2]. It is special, as it is not related to software
verification. The formalization of *characteristic functions* forced us to generalize the
Lebesgue integral, which led to the formalization of Bochner integration. The theorem is
stated in terms of weak convergence on measures, which resulted in many measurability
rules for analytical functions and predicates.
Density Compiler  Bhat et al. [4] describe a density compiler for a simple probabilistic programming language. It processes a program $P$ containing random values into a term describing the density of the results of $P$. Eberl [8] formalized this in Isabelle. The semantics of the programming language is defined using the Giry monad. The distributions of the random values, e.g. uniform, exponential, and normal distribution, were already available.

Markov Processes  The construction and analysis of Markov processes is one of the main applications of Isabelle’s probability theory. Currently we only support discrete-state discrete-time time-homogeneous Markov chains and Markov decision processes. The state space can be countably infinite. To analyse Markov chains and to classify their states, we provide theorems like the Chapman-Kolmogorov equations or the existence of a stationary distribution.

Markov processes are either represented by their trace space $T$, or by their kernel $K$ (i.e. the transition matrix). The kernel of a Markov chain is specified as a function $K = S \rightarrow \mathcal{D}(S)$, and for a Markov decision process as a function $K = S \rightarrow \mathcal{P}(\mathcal{D}(S))$, where $\mathcal{D}(S)$ are all pmf’s on the state set $S$. For Markov chains we can show that the trace space is the only solution to the fixpoint equation:

$$\forall s. \ T_s = \text{do} \{ t \rightarrow K_s; \omega \rightarrow T_t; \text{return} (t \cdot \omega) \} .$$

Where $T_s$ is the trace space with initial state $s$ and $K_s$ is the transition distribution when in state $s$. The do-notation is syntactic sugar for the bind-operator of the Giry monad.

Using this construction we formalized the analysis of expected runtime of the address handshake in the zero-conf protocol and the leaked information in the CROWDS protocol [12], and a framework for probabilistic non-interference [21]. The formalization also allows us to verify algorithms operating on Markov chains and Markov decision processes. We formalized pCTL model checking on Markov chains [13] and most of the pCTL model checking on Markov decision processes. For Isabelle it is not only interesting to verify the algorithms, but also to find ways to certify the results of a model checking run. We can certify exact results, but it is not clear how approximate results can be efficiently generated and certified.

4 Related Work

Hurd [14] formalized probability theory used in the context of program verification. While he formalized measure theory (e.g. Caratheodory’s extension theorem), the focus of his library were discrete random variables. Hurd et al. [15], Audebaud and Paulin-Mohring [1], and Cock [6], did away with the measure theoretic approach, and completely focused on discrete random variables.

Richter [22] formalized the Lebesgue integral in Isabelle, limited to integrable non-negative functions. Wang et al. [23] formalized Caratheodory’s extension theorem in Isabelle, unfortunately their work is not publicly available. Berg et al. [3] used Isabelle to formalize probabilistic semantics, they left proof holes (i.e. sorry statements) which could be closed with our development of probability theory.

Liu et al. [19] formalized the classification of Markov chain states. Their approach is limited to finite-state MCs, and they do not give a construction for Markov chains.
Mhamdi et al. [20] developed Lebesgue integration, but their final goal, entropy, is again limited to finite random variables.

Lester [18] formalized different aspects of measure theory in PVS.

References


Continuous Systems Reachability using Adaptive Runge-Kutta Methods – Formally Verified

Fabian Immler (immler@in.tum.de)

Institut für Informatik, Technische Universität München, Germany

1 Introduction

The work presented in this extended abstract is largely based on the author’s original publications [14, 15].

Many real-world systems with continuous dynamics can be modeled with ordinary differential equations (ODEs). An important task is to determine for a set of initial states all reachable states. This requires to compute enclosures for solutions of ODEs, which is done by tools for guaranteed integration (e.g., by a family of tools surveyed by Nedialkov [17], Nedialkov’s state-of-the-art tool VNODE-LP [18], or COSY [3]) and also by tools for reachability analysis of hybrid systems (with the state-of-the-art tool for linear dynamics SpaceEx [8] and tools supporting non-linear dynamics like flow* [6], HySAT/iSAT [9], or Ariadne [2]).

Such tools aim at computing safe overapproximations, an intended use is often the analysis of safety-critical systems. Therefore any effort to improve the level of rigor is valuable, and such efforts have been undertaken already: Nedialkov [18] implemented VNODE-LP using literate programming such that correctness of the code can be examined by human experts. Taylor models, which are used to represent reachable sets in COSY, flow*, and Ariadne, have been formalized in theorem provers in the context of Ariadne [7] but also as a generic means for validated numerics [5, 20].

Here we present the formal verification of an algorithm for reachability analysis of continuous systems. The algorithm splits, reduces and collects reachable sets during the analysis, crucial features for being able to analyze chaotic systems. Propagation of reachable sets is implemented using higher-order Runge-Kutta methods with adaptive step size control. The formal verification of all those algorithms is a novel contribution and a qualitative improvement on the level of trust that can be put into reachability analysis of continuous systems. Experiments show that our algorithms allow to analyze low-dimensional, non-linear systems that are out of reach for many of the existing tools. Nevertheless, our work should not be considered a rival to the existing tools or concepts, which are more mature and flexible. We would rather like to demonstrate that formal verification does not exclude competitive performance.

*Supported by DFG RTG 1480 (PUMA)
We build on our formalization of affine arithmetic and the Euler method [13]. The verification is carried out with respect to the theory of ODEs in the interactive theorem prover Isabelle/HOL [19].

2 Reachability Analysis

We consider the problem of computing reachable sets for systems defined by an autonomous ODE \( \dot{x} = f(x) \) with \( f : \mathbb{R}^n \to \mathbb{R}^n \). We denote the solution depending on initial condition \( x_0 \) and time \( t \) with \( \varphi(x_0, t) \). Reachability analysis aims at computing (or overapproximating) all states of the system that are reachable from some set of initial states \( X_0 \subseteq \mathbb{R}^n \) within a time horizon \( T \subseteq \mathbb{R} \), i.e., the set \( \varphi(X_0, T) \).

In the following, we illustrate the main ingredients of our algorithm for reachability analysis. We do not claim originality for those ideas, however combining all of them for numerically solving ODEs and especially formally verifying them is, to the best of our knowledge, a novel contribution.

Rigorous Numerics. First of all, in any numerical computation, continuous, real-valued quantities are approximated with finite precision. One therefore needs to cope with round-off errors. Reasoning about them explicitly gets very tedious. We therefore take the approach of set-based computing, or rigorous numerics: The idea is to compute with sets instead of single values and abstract all kinds of errors (including round-off) by including them into the set. The data structure we choose is affine forms, they represent sets called zonotopes and have been successfully applied in hybrid systems analysis [1, 10], but also as numerical domain in static analysis [12].

Guaranteed Runge-Kutta Methods with Step Size Adaptation Bouissou et al. [4] presented the idea to turn “classical” numerical algorithms into guaranteed methods by using affine arithmetic. They illustrated their approach on a stiff (i.e., numerical approximations requiring very small step sizes in parts of the state space) ODE, which makes adaptive step size control necessary. In general, automatic step size adaptation improves the performance of any numerical method, as it avoids wasting computational time on “easy” parts of the solution and maintains high accuracy on “hard” parts of the solution.

Splitting Zonotopes are convex sets, this leads to loss of precision when non-convex sets need to be enclosed. But non-linear dynamics produce non-convex sets, which is why a purely zonotope based approach is likely to fail because of more and more increasing overapproximations. The immediate approach is to split the sets before they grow too large, and have the union of smaller sets represent the larger non-convex set.

Reduction While splitting sets allows to maintain precision in the presence of non-convex sets, it leads to problems when the dynamics produce large sets. Especially when analyzing chaotic systems, small initial sets expand rapidly – due to the dynamics of the system, not necessarily because of inaccurate computations. This may produce a prohibitively large number of split sets. Any possibility to reduce the size of reachable
sets therefore is a valuable improvement because it helps to reduce the number of sets. Our method is based on the idea that whenever a reachable set flows through a hyperplane, it can be reduced to the intersection with that hyperplane.

To this end, we have implemented a functional algorithm to compute the zonotope/hyperplane intersection and verified it in Isabelle/HOL. The intersection is performed geometrically, as described by Girard and Le Guernic [11] and is similar to convex hull algorithms. Such algorithms have been successfully verified with Knuth’s [16] theory of counterclockwise systems for discrete sets of points. As Zonotopes represent continuous sets, we needed to extend Knuth’s theory to continuous vector spaces. The interesting fact is that we combine a mixture of different fields: a discrete geometrical algorithm to perform operations on the continuous sets represented by zonotopes.

References


Generating Abstract Graph-Based Procedure Summaries for Pointer Programs

Christina Jansen (christina.jansen@cs.rwth-aachen.de)

Software Modeling and Verification Group, RWTH Aachen University, Germany

Dynamic data structures such as lists and trees, implemented using pointers, are heavily used in e.g. all kinds of application software, but also device drivers, operating systems, and so forth. While pointers offer a flexible concept allowing for very complex program behaviour, pointer programming is error-prone, even being one of the most common sources of bugs in software [1]. Typical problems are dereferencing of null pointers, creation of memory leaks, unsought aliasing effects and the accidental invalidation of data structures through destructive updates, i.e., errors which are usually difficult to trace. However, with the flexibility of pointer programs comes the complexity of analysing them, as they generally induce unbounded state spaces. A common approach to tackle this problem is to apply abstraction techniques to obtain a finite representation.

In Sect. 1 we present an abstraction technique based on graph grammars for analysing intraprocedural pointer programs and tailor it towards a modular analysis that can handle interprocedural programs in Sect. 2. Sect. 3 discusses analysis internals. In Sect. 4 and 5 we conclude with related work and future research.

1 Heap Abstraction using Graph Grammars

Hyperedge replacement grammars (HRGs) have proven to be an intuitive concept for defining and implementing abstractions [5]. The key idea is to represent heaps as hypergraphs (HGs) that contain placeholders representing abstracted fragments of a data structure. Thus they combine concrete and abstract heap parts. Placeholders are realised by deploying hyperedges, i.e., edges that connect an arbitrary number of nodes, labelled with nonterminal symbols (NTs).

Example 1 A typical implementation of a doubly-linked list consists of a sequence of list elements connected by next (n) and previous (p) pointers. Fig. 1 depicts a HG representation of a such a list. The three nodes (circles) represent objects on the heap. The L-labelled box represents an NT edge of rank two indicating an abstracted doubly-linked list between the first and second attached node. The head-labelled box represents a program variable head referencing the first list element. The connections between NT edges and nodes are labelled with their ordinal. A shaded circle indicates a special type of node,

*Supported by Thomas Noll, Software Modeling and Verification Group, RWTH Aachen University.
called external node, identifiable through its node label. Intuitively these nodes form the coupling links of hypergraphs, defining how they can be attached to each other. For the sake of readability, edges representing selectors (n and p) are depicted as directed edges.

The abstract structures represented by nonterminal labelled hyperedges are specified by HRGs. HRGs consist of a set of production rules, given by a nonterminal as left-hand side and an HG as right-hand side. Intuitively, an HRG can be understood as a connector system. The right-hand sides of the production rules provide the building blocks of the system.

Building blocks are joined to an existing structure, i.e. an HG, by applying production rules, that is, by replacing a nonterminal-labelled hyperedge by a right-hand side graph: The coupling pins of the right-hand side, i.e. the external nodes of the graph, are connected to the socket of the edge to-be-replaced, that is its attached nodes.

Example 2 Fig. 2 specifies an HRG for doubly-linked lists. It employs one NT $L$ and two production rules. The right one recursively adds one list element, whereas the left one terminates a derivation. An example application of the second rule to an HG representing a doubly-linked list is illustrated in Fig. 3.

Abstraction and its reverse operation, concretisation, are implemented by applying grammar rules in forward and backward direction, respectively. Note that, provided the employed HRG captures the data structures arising during program execution (the approach is robust against finitely many heap structures that violate it), the abstract execution terminates and yields a finite state space.

In [4] we have shown how the HRG formalism can be employed for handling Java bytecode with (recursive) methods and local variables. The approach is based on the explicit modelling of the runtime stack on the heap, where unbounded recursion is dealt with by abstraction using HRGs. However, this requires the development of appropriate HRGs that capture not only the data structures arising during program execution, but additionally their interaction with the runtime stack, leading to large and intricate grammar specifications. The situation becomes even more complex when tackling the extension to concurrent Java or dynamic thread generation in general, where the list of threads has to be added to the heap representation as a third component.
2 Summarising Procedure Effects

To overcome this increasing complexity in HRG specification, we advocate an alternative approach, which clearly separates heap abstraction from control abstraction. The former still employs HRGs to handle the data structures occurring during program execution. For dealing with the program code, modular reasoning on the level of procedures is used. More exactly, the goal is to automatically derive a summary, referred to as contract, for each procedure in the given program, which abstractly and comprehensively represents the possible effects of its execution. In essence, a contract specifies a graph transformation in the form of a pre- and a postcondition, capturing the heap state before executing the procedure and after. Preconditions are given by hypergraphs and describe the heap upon procedure entry. Postconditions are represented as sets of hypergraphs associated with the respective precondition that describe the possible changes in the precondition after procedure execution.

Example 3 In Fig. 4, the contract of a simple list reversal procedure is given. It consists of several pre-postcondition-pairs as indicated by the dots, where one is provided in detail. The precondition of this pair describes the situation where the reversal procedure is entered with a variable head set to the first and tail to the last element of a doubly-linked list. The postcondition is a singleton set stating that the doubly-linked list from head to tail is reversed, indicated by the mirrored L-labelled hyperedge. The contracts indicated by the dots comprise e.g. the situation where the reversal algorithm is called on an empty list.

Note that restricting the precondition to the fragment reachable from the procedure parameters is sufficient for capturing the procedure effect as the rest of the heap is non-accessible and therefore immutable. This generally improves the modularity of the approach, as the resulting preconditions are applicable to a larger set of initial heaps.

Given a pointer program, our analysis aims to derive sound (and precise) procedure contracts by symbolic execution. As these programs may contain e.g. recursive procedures or looping constructs, the handling of unbounded runs and thus termination of the symbolic execution, while preserving soundness, is of great importance.

3 Deriving Contracts

The proposed approach is based on an interprocedural dataflow analysis (IPA), which handles the runtime stack and local variables. The analysis information derived at each program point is provided as a stack of (sets of) contracts. Each stack entry comprises the effect of the procedure execution up to the current program point.

Example 4 In Fig. 5, a schematic execution of a call to procedure p passing procedure parameters l₁, . . . , lₖ at program point m can be found. The current analysis information in form of a contract stack at the calling site is depicted by the rectangle above of the call,
containing the set of contracts with $d$ as top-most entry. Upon call of procedure $p$, a new contract $e$ is generated using the current heap state provided by $d$ as basis. The precondition of $e$ is the fragment of the current heap reachable from the parameters $l_1, \ldots, l_k$. This contract is pushed onto the existing contract stack of procedure $p$, as indicated by the upper dashed arrow, resulting in the stack $[e, e_0, \ldots]$. The procedure body of $p$ is executed symbolically on the new contract $e$, which is depicted by the wavy edge. After the execution of $p$ terminated, we end up in program state $n$ with analysis information $[e', e_0, \ldots]$. Thus $e'$ summarises the result of $p$'s execution on the initially calculated precondition. This procedure summary is integrated into the top-most contract at the calling site $m$ of $p$ as indicated by the bottom dashed arrow and resulting in the adapted top-most stack entry $d'$.

To make use of the modularity offered by contracts and to ensure that symbolic execution eventually terminates, we reuse contracts whenever possible. That is, whenever the analysis reaches a procedure call, it first checks if one of the previously generated contracts coincides with the current heap state. If so, the contract is applied directly without recomputation of the procedure body. Otherwise a new contract for that situation is generated and we proceed with the contract derivation as illustrated in the previous example.

We utilise a demand-driven IPA based on fixpoint iteration to determine the least set of contract information at every program point. Thus, the $n$-th stack entry results from the $n$-th iteration of the analysis. This approach forms an instance of the general IPA framework introduced in [6]. That is, provided the abstract state space of the program is finite, termination of the program analysis is guaranteed and the results coincide with the meet-over-all-paths solution according to the Interprocedural Coincidence Theorem [6]. Thus the approach tackles the “important, and still open, problem of handling an unbounded number of live cutpoints under abstraction” [7] successfully.

4 Related Work

In [10] the runtime stack is explicitly represented and abstracted as a linked list, using shape analysis. This is similar to our previous work in [4]. The alternative interprocedural heap abstraction approach developed in the present paper is based on the general IPA framework as described, e.g., in [6,13]. Specific instances have been proposed for the finite, distributive subset case (IDFS; [9]) and the distributive environments case (IDE; [12]).
A generalisation of these is presented in [14] where a class of abstract domains and associated transformations is defined which allows to obtain precise and concise procedure summaries. However, these instances are not applicable in our setting due to the combination of recursion and local variables. A framework for interprocedural heap shape analysis in the cutpoint-free case is first proposed in [11] and later generalised in [7] by admitting non-live cutpoints. Moreover, [2] describes a modular interprocedural shape analysis that can handle a bounded number of cutpoints (interpreted as additional procedure parameters). However, the analysis is restricted to the setting of singly-linked lists, while the approach proposed here deals with all data structures of bounded tree-width.

5 Conclusion

This paper presents a novel IPA for automatically deriving procedure contracts for pointer programs. The IPA builds upon an abstraction framework based on HRGs [5]. We follow an approach that separates heap from control abstraction. The proposed analysis summarises procedure effects in so-called contracts and thus allows a modular reasoning at procedure level. It supports recursive procedures with local variables and cutpoints, i.e., heap objects that are shared between the heap fragment a procedure call operates on and the calling context. It turns out that the HRG approach is particularly suited for determining and applying such contracts as it offers an intuitive formalism for describing heap transformations, which moreover can be automatically derived.

We are planning to extend our techniques to concurrent programs with threads. In this setting, we expect the contract approach to be even more beneficial than in the interprocedural case. While the latter can alternatively be handled by providing HRG rules that allow to abstract the runtime stack in order to get a finite-state representation [4], it seems hopeless to develop similar (efficient) abstraction techniques for the concurrent case [8]. Our idea to overcome the efficiency problem is to develop a thread-modular analysis that, similarly to [3], avoids the enumeration of all possible interleavings between threads.

References


A Greedy Approach for the Efficient Repair of Stochastic Models

Nils Jansen
RWTH Aachen University, Germany

Abstract. For discrete-time probabilistic models there are efficient methods to check whether they satisfy certain properties. If a property is refuted, available techniques can be used to explain the failure in form of a counterexample. However, there are no scalable approaches to repair a model, i.e., to modify it with respect to certain side conditions such that the property is satisfied. In this paper we propose such a method, which avoids expensive computations and is therefore applicable to large models. A prototype implementation is used to demonstrate the applicability and scalability of our technique.

Discrete-time Markov chains (DTMCs) are a widely used modeling formalism for systems that exhibit probabilistic behavior, some typical application areas being distributed computing, security, hardware, and systems biology. DTMCs can be seen as directed graphs whose transitions are equipped with probabilities. This enables the possibility to use well-known graph algorithms such as Dijkstra’s shortest path search [1] for the investigation of properties on DTMCs. A popular language to specify properties of such models is probabilistic computation tree logic (PCTL) [2] which is the probabilistic extension of the well-known computation tree logic (CTL).

Model checking of an important subclass of PCTL properties or \(\omega\)-regular properties can be reduced to model checking reachability problems. In particular, it is verified whether the probabilities of reaching a set of distinguished target states are within some required thresholds. Efficient probabilistic model checkers like PRISM [3] or MRMC [4] are available.

In context of these systems, a natural question is how to correct or identify errors in a model of a safety-critical system. In the recent past, much effort has been made in automatically generating explanations for the failure of a property in the form of counterexamples. For an overview on different approaches and literature we refer to [5]. In spite of various efficient methods for counterexample generation, a still open problem is how to automatically repair a DTMC model that does not meet a certain requirement.

A first approach, referred to as model repair for DTMCs, was presented in [6]. Basically, linear combinations of real-valued parameters are added to the transition probabilities of a DTMC that does not satisfy a desired reachability

⋆ This work was partially supported by the Excellence Initiative of the German federal and state government.
property. Additionally, a cost-function over the parameters is given. The goal is to find an evaluation of the parameters which on the one hand induces the satisfaction of the property and on the other hand minimizes the value of the cost-function, i.e., changing the transition probabilities and thereby repairing the DTMC under minimal costs.

Formally, the underlying model is a parametric discrete-time Markov chain (PDTMC). Such models can also be used in early system development stages, where the parameters represent design variables whose values can be fixed later. For verifying this parametric model, the values shall then be fixed such that the resulting instantiated model satisfies some properties within a fixed probability range and is optimal (or nearly optimal) with respect to a given objective function. Recently, some approaches were proposed to represent the probability that a PDTMC satisfies a required property in form of a rational function over the parameters [7, 8], as being implemented in the tool PARAM [9]. Optimization according to the objective function under the given side conditions can be used to synthesize suitable parameter values. In [6], such a rational function is computed for the PDTMC underlying the model repair problem. Then a non-linear optimization problem [10] is solved implying that the desired property is satisfied for this formula while the cost-function is minimized. This can be done, e.g., via IPOPT [11]. If satisfiable, the resulting evaluation is a solution for the model repair problem. Moreover, a method for Markov decision processes (MDPs) was proposed, encompassing approximative methods [12]. Model repair for non-stochastic system has, e.g., been studied in [13].

The main practical obstacle of using non-linear optimization, be it using a dedicated optimization algorithm or using an SMT-solver for non-linear real algebra [14] coupled with a binary search towards the optimal solution, is scalability. As even the computation of the rational function involves costly computations of greatest common divisors of polynomials, approaches like [7, 8] are inherently restricted to small PDTMCs with just a few parameters. Thus there is a need for scalable model repair methods, avoiding the involvement of state-of-the-art PDTMC verification techniques.

Here, we present a new technique which we call local repair. We define three subclasses of PDTMC models with increasing expressivity. For each class, we present methods that start from an initial parameter assignment and iteratively change the parameter values by local repair steps. Intuitively, the three classes are defined as follows:

**Type-1:** PDTMCs where each variable appears on at most one transition.

**Type-2:** PDTMCs where each variable appears in at most one probability distribution, i.e., variables can occur on several transitions which all emerge from the same state.

**Type-3:** General PDTMCs, allowing each variable to appear several times possibly in different distributions.

*Example 1.* Figure 1 shows examples for the three PDTMC classes. Note that sometimes Type-2 PDTMCs can be transformed to Type-1 PDTMCs.
As mentioned before, finding an evaluation of parameters that satisfies a property with respect to the imposed restrictions, is hard. We therefore aim at defining a greedy method to stepwise improve a given initial valuation. More precisely, given an initial valid valuation for a PDTMC, our goal is to iteratively manipulate the valuation such that in the induced DTMC the probability of reaching target states is successively changed until the property is met.

For our approach, correctness and completeness can be shown in the sense that each local repair step improves the reachability probability towards a desired bound for a repairable PDTMC and terminates with an optimal solution.

Example 2. Consider the PDTMC $P$ depicted in Figure 2. State $(2,2)$ shall be the target state and $(1,2)$ shall be the initial state. Using computations as described in [7, 8], the probability to reach the target state from the initial state is described by the rational function $p(1,2) = \frac{-x+y+1}{-x+y+2}$. In this simple example this function is linear, however, this is no necessarily the case for real applications.

Using an initial valuation $v$ of the parameters $x$ and $y$ with $v(x) = v(y) = 0$ yields the DTMC $D$ depicted in Figure 2. Using these values to evaluate the rational function yields a reachability probability of $\frac{1}{2}$ which is the probability to reach state $(2,2)$ from $(1,2)$ inside this DTMC. We want to reduce this probability via model repair.

Consider the evaluation $\hat{v}$ with $\hat{v}(x) = 0.2, \hat{v}(y) = 0$. The resulting DTMC $\hat{D}$ is depicted in Figure 2 on the right. The probability to reach state $(2,2)$ from state $(1,2)$ is now $\hat{p}(2,1) = \frac{1}{3} < \frac{1}{2}$. Thus, the $\hat{D}$ is the repaired DTMC with respect to the PDTMC $P$.

We implemented our approach in a prototype and tested it thoroughly in an application from a robotics scenario, where the given environment is modeled by a Markov Decision Process (MDP) and where a controller—modeled by a DTMC—is synthesized via reinforcement-learning [15]. This controller shall be repaired until a certain property is satisfied. Furthermore, we present well-known benchmarks from the PRISM benchmark suite and categorize each of them into one of our three PDTMC subclasses. The experiments show the feasibility of our approaches, where the standard method as proposed in [6] immediately fails even for very small systems.
Fig. 2. PDTMC $\mathcal{P}$, DTMC $D$, and repaired DTMC $\hat{D}$.

References

Analyzing Expected Outcomes and (Positive) Almost–Sure Termination of Probabilistic Programs is Hard

Benjamin Lucien Kaminski†(benjamin.kaminski@cs.rwth-aachen.de)

Software Modeling and Verification Group, RWTH Aachen, Germany

Probabilistic programs [1] are imperative sequential programs with the ability to toss a biased coin and proceed their computations depending on the outcome of the coin toss. They are used in security to describe cryptographic constructions (such as randomized encryption) and security experiments [2], in machine learning to describe distribution functions that are analyzed using Bayesian inference [3], and in randomized algorithms. They are typically just a small number of lines, but hard to understand and analyze, let alone algorithmically. We consider two major analysis problems for these programs:

1. Computing expected outcomes: Is the expected outcome of a program (variable) smaller, equal, or larger than a given rational number?

2. Deciding (positive) almost–sure termination [4]: Does a program terminate with probability one (in an expected finite number of computation steps)?

Regarding almost–sure termination, the majority of the literature does either not consider the hardness of the problem or states that it must be somewhat harder to decide than the classical termination problem, as arithmetical instead of topological reasoning is needed (see e.g. [5, 6]).

In our work, we strive to give a precise classification of the computational and arithmetical complexity of solving the aforementioned analysis problems. Most of the results summarized here together with proofs can be found in [7].

Preliminaries: Our classifications will be in terms of levels in the arithmetical hierarchy [8]—a concept which we first briefly recall: For any \( n \in \mathbb{N} \) the class \( \Sigma^0_n \) is defined as \( \Sigma^0_n = \{ A | A = \{ \vec{x} \mid \exists y_1 \forall y_2 \exists y_3 \cdots \exists /\forall y_n : (\vec{x}, y_1, y_2, y_3, \ldots, y_n) \in R \} \}, R \) is a decidable relation}, the class \( \Pi^0_n \) is defined as \( \Pi^0_n = \{ A | A = \{ \vec{x} \mid \forall y_1 \exists y_2 \forall y_3 \cdots \exists /\forall y_n : (\vec{x}, y_1, y_2, y_3, \ldots, y_n) \in R \} \}, R \) is a decidable relation}, and the class \( \Delta^0_n \) is defined as \( \Delta^0_n = \Sigma^0_n \cap \Pi^0_n \). We implicitly always quantify over \( \mathbb{Q}^+ \) and by the boldface \( \vec{x} \)'s we mean tuples over \( \mathbb{Q}^+ \). In a formula, multiple consecutive quantifiers of the same type can be...

*This research is funded by the Excellence Initiative of the German federal and state governments and by the EU FP7 MEALS project.
†I would like to thank Joost-Pieter Katoen and Luis María Ferrer Fioriti for many valuable discussions and comments.
contracted to one quantifier of that type, so \( n \) really refers to the number of quantifier
alternations rather than to the number of quantifiers actually used in a formula. A set
\( \mathcal{A} \) is called arithmetical, iff \( \mathcal{A} \in \Gamma_0^n \), for some \( \Gamma \in \{ \Sigma, \Pi, \Delta \} \) and some \( n \in \mathbb{N} \). The
arithmetical sets form a strict hierarchy, i.e. \( \Delta_0^n \subseteq \Sigma_0^n, \Pi_0^n \subseteq \Delta_{n+1}^0 \) for all \( n \geq 1 \). We have
that \( \Sigma_0^0 = \Pi_0^0 = \Delta_0^0 = \Delta_0^1 \) is precisely the class of decidable sets and \( \Sigma_1^0 \) is precisely the
class of recursively enumerable sets.

Next we recall the concept of many–one reducibility and the concept of completeness
[8]: Let \( \mathcal{A}, \mathcal{B} \) be arithmetical sets and let \( X \) be some appropriate universe, such that
\( \mathcal{A}, \mathcal{B} \subseteq X \). \( \mathcal{B} \) is called many–one–reducible to \( \mathcal{A} \), denoted \( \mathcal{B} \leq_m \mathcal{A} \), iff there exists a
computable function \( f : X \to X \), such that \( \forall \bar{x} \in X : (\bar{x} \in \mathcal{B} \iff f(\bar{x}) \in \mathcal{A}) \). Intuitively
this means that it is computationally at least as hard to solve problem \( \mathcal{A} \) as it is to solve
problem \( \mathcal{B} \). \( \mathcal{A} \) is called \( \Gamma_0^n \text{–complete} \), for \( \Gamma \in \{ \Sigma, \Pi, \Delta \} \), iff both \( \mathcal{A} \in \Gamma_0^n \) and \( \mathcal{A} \) is \( \Gamma_0^n \text{–hard} \),
meaning \( \mathcal{B} \leq_m \mathcal{A} \), for any set \( \mathcal{B} \in \Gamma_0^n \). The universal halting problem (does a classical,
i.e. non–probabilistic, program terminate on any given input?), denoted \( \overline{UH} \), for instance,
is \( \Pi_0^n \text{–complete} \) whereas its complement, denoted \( \overline{\overline{UH}} \), is \( \Sigma_2^n \text{–complete} \) [9].

Notice that if \( \mathcal{B} \) is \( \Gamma_0^n \text{–complete} \) and \( \mathcal{B} \leq_m \mathcal{A} \) then \( \mathcal{A} \) is \( \Gamma_0^n \text{–hard} \). Another important
fact about \( \Sigma_0^n \text{–} \) and \( \Pi_0^n \text{–complete} \) sets is that they are in some sense the most complicated
sets in \( \Sigma_0^n \) and \( \Pi_0^n \), respectively. Formally, this can be expressed as follows: If \( \mathcal{A} \) is \( \Sigma_0^n \text{–complete} \), then \( \mathcal{A} \in \Sigma_0^n \setminus \Pi_0^n \). Analogously if \( \mathcal{A} \) is \( \Pi_0^n \text{–complete} \), then \( \mathcal{A} \in \Pi_0^n \setminus \Sigma_0^n \). This
implies in particular that if \( \mathcal{A} \) is \( \Sigma_0^n \text{–complete} \) or \( \Pi_0^n \text{–complete} \) then \( \mathcal{A} \not\in \Delta_0^n \).

**Probabilistic Programs:** We study analysis problems for pGCL–like probabilistic pro-
grams à la McIver & Morgan [10]—an extension of Dijkstra’s GCL programs [11]. If \( v \)
stands for a program variable, \( e \) for an arithmetical expression over program variables,
and \( p \in [0, 1] \subset \mathbb{Q} \) for a probability, then the programs that we consider adhere to the
following grammar:

\[
P \quad \longrightarrow \quad v := e \mid \ P \cdot P \mid \{P\} [p] \{P\} \mid \text{WHILE} (b) \{P\}
\]

While assignment, concatenation, and the while–loop are classical programming language
constructs, the command \( \{P_1\} [p] \{P_2\} \) represents a probabilistic choice between the two
programs \( P_1 \) and \( P_2 \). With probability \( p \) the program \( P_1 \) is executed, while with proba-
bility \( 1 – p \) the program \( P_2 \) is executed. While classical programs upon termination yield
a variable valuation, probabilistic programs instead yield a subdistribution over variable
valuations where the missing probability mass is the probability of non–termination. For
example, the program

\[
x := 0; \{c := 0\} \{1/2\} \{c := 1\}; \text{WHILE} (c=1) \{x := x+1; \{c := 0\} \{1/2\} \{c := 1\}\}
\]

terminates with probability 1 and establishes for \( c \) the valuation 0 with probability 1 and
for \( x \) a geometric distribution over the valuations 0, 1, 2, \ldots

**Expected Outcomes:** The first analysis problem we consider is to approximate ex-
pected outcomes. By the expected outcome of a variable \( v \) we mean the value we expect
\( v \) to have after the program has been executed. In the above example program, for in-
stance, the expected outcome of \( x \) is 1 as \( 1/2 \cdot 0 + 1/4 \cdot 1 + 1/8 \cdot 2 + \cdots = 1 \). Formally, the
expected outcome of a variable \( v \) after executing the program \( P \) starting with variable valuation \( \eta \), denoted \( \text{E}_{P,\eta}(v) \), can be expressed by

\[
\text{E}_{P,\eta}(v) = \sum_{i=1}^{\infty} \sum_{j=0}^{\infty} W(P, \eta, v, i, h(j)),
\]

where \( W \) is a computable function that simulates the program \( P \) on \( \eta \) for \( i \) computation steps while resolving all probabilistic choices that occur during simulation according to a sequence \( h(j) \in \{L, R\}^* \) which encodes whether to take the Left or the Right branch when the simulation encounters a probabilistic choice. Note that a computable enumeration \( h \) of all words over \( \{L, R\} \) exists. If \( P \) terminates after exactly \( i \) computation steps with exactly \( |h(j)| \) probabilistic choices, then \( W \) returns the resulting valuation of the variable \( v \) multiplied with the probability with which the probabilistic choices encoded in \( h(j) \) would have been made, otherwise 0. Such a function \( W \) can be obtained by a slight adaption of the Kleene Normal Form Theorem to probabilistic programs [12]. With the above formula we sum over all possible execution steps and all possible resolutions of the probabilistic choices and sum up all resulting valuations of \( v \) weighted with their according probabilities.

For approximations of expected outcomes from below, we define the set \( \mathcal{LEX}\mathcal{P} \) by

\[
(P, \eta, v, q) \in \mathcal{LEX}\mathcal{P} \iff q < \text{E}_{P,\eta}(v).
\]

A tuple \( (P, \eta, v, q) \) is in \( \mathcal{LEX}\mathcal{P} \) if the expected outcome of variable \( v \) after executing \( P \) starting in \( \eta \) is strictly larger than the rational \( q \). This set can be defined by a \( \Sigma^0_1 \)–formula, namely

\[
(P, \eta, v, q) \in \mathcal{LEX}\mathcal{P} \iff \exists i \exists j \colon q < \sum_{i=1}^{\infty} \sum_{j=0}^{\infty} W(P, \eta, v, i, h(j)),
\]

and is therefore recursively enumerable. This means that arbitrarily close rational approximations from below can effectively be enumerated.

For approximations from above the set \( \mathcal{REX}\mathcal{P} \) is defined by \( (P, \eta, v, q) \in \mathcal{REX}\mathcal{P} \) iff \( q > \text{E}_{P,\eta}(v) \). If \( \mathcal{REX}\mathcal{P} \) was recursively enumerable, arbitrarily close rational approximations from below and from above could effectively be enumerated and expected outcomes would therefore fall into the class of computable reals. We can, however, prove that \( \mathcal{REX}\mathcal{P} \) is not only in \( \Sigma^0_2 \), defined by

\[
(P, \eta, v, q) \in \mathcal{REX}\mathcal{P} \iff \exists \delta \forall i \forall j \colon q - \delta > \sum_{i=1}^{\infty} \sum_{j=0}^{\infty} W(P, \eta, v, i, h(j)),
\]

but is also \( \Sigma^0_2 \)–complete. We can prove the \( \Sigma^0_2 \)–hardness of \( \mathcal{REX}\mathcal{P} \) by showing \( \mathcal{UH} \leq_m \mathcal{REX}\mathcal{P} \). The set \( \mathcal{REX}\mathcal{P} \) would be recursively enumerable, if there was access to an oracle for the halting problem (does a classical program halt on a certain fixed input?) [8].

Deciding whether some rational \( q \) equals the expected outcome of \( v \) is even harder: The according set, denoted \( \mathcal{EX}\mathcal{P} \), is defined as \( (P, \eta, v, q) \in \mathcal{EX}\mathcal{P} \) iff \( q = \text{E}_{P,\eta}(v) \). This set is in \( \Pi^0_2 \). To see this, consider that \( (P, \eta, v, q) \in \mathcal{EX}\mathcal{P} \) means that both \( (P, \eta, v, q) \notin \mathcal{LEX}\mathcal{P} \) and \( (P, \eta, v, q) \notin \mathcal{REX}\mathcal{P} \). By negating the defining formulas for \( \mathcal{LEX}\mathcal{P} \) and \( \mathcal{REX}\mathcal{P} \) we obtain a \( \Pi^0_1 \)– and a \( \Pi^0_2 \)–formula, respectively. The conjunction of these two formulas gives a \( \Pi^0_2 \)–formula defining \( \mathcal{EX}\mathcal{P} \). By showing \( \mathcal{UH} \leq_m \mathcal{EX}\mathcal{P} \) we can also prove the \( \Pi^0_2 \)–hardness of \( \mathcal{EX}\mathcal{P} \) and therefore we can establish that \( \mathcal{EX}\mathcal{P} \) is \( \Pi^0_2 \)–complete. This means
that determining exact expected outcomes is \textit{not semi–decidable, even if there was access to an oracle for the halting problem.}

\textbf{Almost–Sure Termination:} The probability that $P$ terminates on input $\eta$, denoted $\Pr_{P,\eta}(\downarrow)$, can be expressed by

$$
\Pr_{P,\eta}(\downarrow) = \sum_{i=1}^{\infty} \sum_{j=0}^{\infty} T(P, \eta, i, h(j)) ,
$$

where $T$ is a computable function similar to $W$, but instead of returning a variable valuation multiplied with the probability of the probabilistic choices encoded in $h(j)$, $T$ returns only those probabilities. We say that a program $P$ \textit{terminates almost–surely} on input $\eta$ iff $\Pr_{P,\eta}(\downarrow) = 1$. The according set $\text{AST}$ is defined as $(P, \eta) \in \text{AST}$ iff $\Pr_{P,\eta}(\downarrow) = 1$. By showing $\text{AST} \leq_m \text{EXP}$ we can prove that $\text{AST} \in \Pi_2^0$ and by showing $\text{UH} \leq_m \text{AST}$ we can establish that $\text{AST}$ is also $\Pi_2^0$–hard and hence $\Pi_2^0$–complete.

For ordinary programs, we have that the universal halting problem is $\Pi_2^0$–complete, whereas the non–universal version of the halting problem is only $\Sigma_1^0$–complete, thus computationally less hard to solve. Interestingly, for almost–sure termination this is not the case: The set of all universally almost–surely terminating programs, denoted $\text{UAST}$, is defined by $P \in \text{UAST}$ iff $\forall \eta: \Pr_{P,\eta}(\downarrow) = 1$. By showing $\text{AST} \leq_m \text{UAST}$ we can prove that $\text{UAST}$ is $\Pi_2^0$–hard. Furthermore, we can express $\text{UAST}$ using $\text{AST}$, namely by

$$
P \in \text{UAST} \iff \forall \eta: (P, \eta) \in \text{AST} ,
$$

thus $\text{UAST}$ is in the universal closure of $\text{AST}$. As $\Pi_2^0$ is closed under universal quantification for any $n \geq 1$ and $\text{AST} \in \Pi_2^0$, we have that $\text{UAST} \in \Pi_2^0$ and therefore $\text{UAST}$ is also $\Pi_2^0$–complete.

\textbf{Positive Almost–Sure Termination:} Along the lines of [13], we can express the expected termination time of $P$ on input $\eta$, denoted $E_{P,\eta}(\downarrow)$, by

$$
E_{P,\eta}(\downarrow) = \sum_{k=1}^{\infty} \left( 1 - \sum_{i=1}^{k-1} \sum_{j=0}^{2^{k-1}-1} T(P, \eta, i, h(j)) \right) .
$$

We say that a program $P$ \textit{terminates positively almost–surely} on input $\eta$ iff $E_{P,\eta}(\downarrow) < \infty$, i.e. the expected number of steps until termination is finite. The according set, denoted $\text{PAST}$, is defined as $(P, \eta) \in \text{PAST}$ iff $E_{P,\eta}(\downarrow) < \infty$. Notice that if the expected number of steps until termination is finite, then the program also terminates almost–surely. However, there exist programs that \textit{do} terminate almost–surely but \textit{not} positively, i.e. we have that $\text{PAST} \subsetneq \text{AST}$. $\text{PAST}$ is computationally more benign than $\text{AST}$, as it can be defined by

$$(P, \eta) \in \text{PAST} \iff \exists c \forall k: \sum_{k=1}^{k} \left( 1 - \sum_{i=1}^{k-1} \sum_{j=0}^{2^{k-1}-1} T(P, \eta, i, h(j)) \right) < c ,$$

160
and we thus have \( \text{PAST} \in \Sigma^0_2 \). We can furthermore show \( UH \leq \text{PAST} \) and therefore establish that \( \text{PAST} \) is \( \Sigma^0_2 \)-complete. The result of this reduction is somewhat counterintuitive as each classical program that does not terminate on some input is transformed by a computable function \( f \) into a probabilistic program that does terminate in an expected finite amount of steps, whereas the same \( f \) transforms a program that does terminate on all inputs into a program that does not terminate in an expected finite amount of steps.

The set corresponding to the universal version of \( \text{PAST} \), denoted \( \text{UPAST} \), is defined by \( P \in \text{UPAST} \) iff \( \forall \eta: (P, \eta) \in \text{PAST} \). As the universal closure of a set in \( \Sigma^0_n \) is in \( \Pi^0_{n+1} \) for any \( n \geq 1 \) and \( \text{PAST} \in \Sigma^0_2 \), we have that \( \text{UPAST} \in \Pi^0_3 \). Furthermore consider the \( \Pi^0_3 \)-complete set \( \text{COF} \) of classical programs defined by \( P \notin \text{COF} \) iff \( P \) does not terminate on infinitely many inputs [9]. By showing \( \text{COF} \leq_m \text{UPAST} \), we can prove the \( \Pi^0_3 \)-completeness of \( \text{UPAST} \) and thus \( \text{UPAST} \) is even harder to solve than \( \text{AST} \).

**References**


161
Two-Player Perfect-Information Shift-Invariant Submixing Stochastic Games Are Half-Positional*

Edon Kelmendi (edon.kelmendi@labri.fr)

LaBRI, Université de Bordeaux, France

The object of this study are two-player zero-sum stochastic games with perfect information and infinite duration, the property of these games that we study is positionality, namely the question: when can one player play optimally without memory and without a source of random bits? We give a sufficient condition on the payoff function to achieve half-positionality (Player 1 can play optimally with a positional strategy). The condition is that the payoff function should be shift-invariant and submixing. In [2] it is shown that one-player stochastic games with shift-invariant and submixing payoff functions are positional, and in [3] the half-positionality of deterministic two-player games with shift-invariant and submixing payoff functions is demonstrated. We extend these two results to half-positionality of two-player stochastic games.

Two player games can be used to model systems where the environment is seen as the adversary and the objective that we want to reach is given by the payoff function. Then the question of synthesizing a controller that reaches the given objective is equivalent to finding an optimal strategy.

A payoff function \( f \) is a bounded and Borel-measurable function mapping plays \( S(AS)^\omega \) to \( \mathbb{R} \). Where \( S \) is the set of states and \( A \) is the set of actions. We make the main result more precise.

**Definition 1.** A payoff function \( f \) is shift-invariant (also known as prefix-independent) if for all finite histories \( h \in S(AS)^* \) and infinite plays \( p \in S(AS)^\omega \),

\[
    f(hp) = f(p).
\]

**Definition 2.** A payoff function \( f \) is submixing if for all plays \( p_1, p_2 \in S(AS)^\omega \), and any shuffling \( p \) of \( p_1 \) and \( p_2 \),

\[
    f(p) \leq \max\{f(p_1), f(p_2)\}.
\]

We say that \( p = u_1v_1u_2v_2\cdots \) is a shuffling of \( p_1 \) and \( p_2 \) if they can be factorized into \( p_1 = u_1u_2\cdots \), \( p_2 = v_1v_2\cdots \) where \( u_i \) and \( v_i \) are words in \( S(AS)^+ \).

*This is joint work with Hugo Gimbert (hugo.gimbert@labri.fr)
The players have opposing objectives since the game is zero-sum, we assume that the objective of Player 1 is to maximize the expected payoff while Player 2 has the opposite objective.

Now the main result can be stated in the following way:

**Theorem 1.** Assume that the payoff function is both shift-invariant and submixing, then Player 1 has an optimal strategy that is both pure and stationary.

In the way of proving our main result we also show that for all $\epsilon > 0$, given that the payoff function is shift-invariant, both players have $\epsilon$-subgame perfect strategies, i.e. strategies that are $\epsilon$-optimal not only from the beginning of the game but also for any finite play already played.

The submixing and shift-invariant payoff functions form a class that is large enough to have the half-positionality of games equipped the following well-known payoff functions follow easily from our main result:

- **Mean payoff**: each state is labeled by some reward $r : S \rightarrow \mathbb{R}$ which for every play induces an infinite sequence of rewards $r_1, r_2, \ldots$. Then the payoff function is
  \[f(r_1r_2\cdots) = \limsup_n \frac{1}{n} \sum_{j=1}^{n} r_j.\]

- **Discounted payoff**: each state is labeled by some immediate reward $r : S \rightarrow \mathbb{R}$ and a discount factor $\lambda : S \rightarrow [0,1)$ and the payoff function measures the long-term performance with an inflation rate:
  \[f(s_1s_2\cdots) = \sum_{j=1}^{\infty} \left(r(s_j) \prod_{k=1}^{j} \lambda(s_k)\right).\]

- **Parity condition**: states are labeled by integers $c : S \rightarrow \mathbb{N}$, Player 1 wins if the largest integer seen infinitely often is odd:
  \[f(c_1c_2\cdots) = \begin{cases} 1 & \text{if } \limsup c_1c_2\cdots \text{ is odd} \\ 0 & \text{otherwise}. \end{cases}\]

- **Limsup (liminf) payoff**: states are labeled by rewards $r : S \rightarrow \mathbb{R}$, and the payoff function computes the limsup (liminf) of the rewards:
  \[f(r_1r_2\cdots) = \limsup (r_1r_2\cdots), \quad \text{or} \quad f(r_1r_2\cdots) = \liminf (r_1r_2\cdots).\]

The (half)-positionality of games equipped with the payoff functions above have been shown by numerous authors, using different techniques. We provide a unified proof in the quest to understand what is the common denominator that allows for very simple optimal strategies in all these seemingly different games. On the other hand the main theorem
Two-player shift-invariant and submixing stochastic games are half-positional

also provides an easy to use tool to check the half-positionality of games, albeit it only is a sufficient condition. One can also construct new types of games that are half-positional, using the main theorem as a guide.

Interesting future work would be to see if the theorem holds for infinite but compact action spaces, or to try to provide a necessary condition for half-positionality.

Given a payoff function such that every one-player stochastic game equipped with it is positional does not imply that the two-player game is half-positional. We provide a counter-example.

References


Traffic Data Dissemination in Realistic Urban VANETs Environment

Saifullah Khan (saifullah.khan@uni-oldenburg.de)

SCARE Research Training Group, University of Oldenburg, Germany

1 Introduction

Vehicular ad-hoc networks (VANETs) using IEEE 802.11p-based technology are expected to support a large spectrum of mobile distributed applications ranging from traffic alert dissemination and dynamic route planning to context-aware advertisement\(^1\) and file sharing. Inter vehicle communication (IVS) is becoming a promising field of research and we are moving closer to the vision of intelligent transportation systems (ITS), which can enable a wide range of applications, such as collision avoidance, emergency message dissemination, dynamic route scheduling, real-time traffic condition monitoring and different kind of infotainment information spreading (i.e., movies, gaming and advertisement). In urban VANETs, more issues should be considered in the design of routing protocols such as the large number of vehicles, various traffic signals, the restricted movement area, uneven vehicle distributions, no transmitting power constraints, obstacles. Among these factors, the impact of obstacles on the communication quality is a more representative characteristic in urban scenario. VANETs are a special type of mobile ad-hoc networks (MANETs), but have several properties that distinguish them, among them relatively high node mobility. This means that, despite safety-criticality of some of the envisioned applications, the probability of network segregation is higher and end-to-end connectivity is not guaranteed. Indeed, the traditional node-centric view of the routes (i.e., an established route is a succession of nodes between the source and destination) leads to frequent route breakage due to node mobility and the ubiquitous signal-weakening obstructions, such as buildings, vegetation, or parked and moving vehicles in the vicinity [1]. In contrast to, e.g., airborne ad-hoc networks (AANETs), blocked line of sight (LOS) in city areas is the rule rather than an exception [2]. To account for the instability of multi-hop wireless communications in vehicular environments, routing protocols exploit geographical information that can be broadly be classified into two categories, namely topology or position based. Multi-hop data delivery through VANETs is complicated by the fact that node densities in vehicular networks are far from even, rapidly fluctuating, and sometimes sparse. To solve the connectivity problem, different types of routing protocols have been presented in the literature, with most of them exploiting geographical information. An important class of protocols selects forwarding paths or nodes based on the current road-traffic conditions, and hence on the potential presence of relaying nodes along the path. Although some proposals to estimate road traffic conditions in situ have been reported in the recent literature, they usually imply a high communication overhead as they collect non-local traffic information. In this context, this paper introduces TASR,

\(^1\)The MIT also provides information online, [http://cartel.csail.mit.edu](http://cartel.csail.mit.edu).
a locally informed multi-hop routing protocol for vehicular ad-hoc networks in city environments. TASR is a fully distributed protocol without the need of any pre-installed infrastructure, just relying on the mobile nodes. Its mechanism is designed to improve the success probability of routing protocols by dynamically estimating the multi-hop forwarding capabilities of road segments and the network connectivity among them.

2 System Model

We assume that vehicles communicate with each other through a short-range wireless channel (300m) and each node can obtain its latest position via GPS or A-GPS. Vehicles enclose their own physical location, current velocity, and direction information in their periodic beacon messages, and this information can be overheard by their one-hop neighbors. We also assume that vehicles are equipped with preloaded digital maps, which provide street-level maps as well as traffic statistics concerning expected traffic density, average vehicle speed on roads at different times of the day, and traffic signal schedule at intersections. The latest one is developed by Map-Mechanics\(^2\) and includes road speed data and an indication of the relative density of vehicles on each road. A three-dimensional box-shaped volume constrains our model region. Within this volume, points represent vehicles and square blocks represents obstacles that shield communication. As shown in Fig1, two points between which an unobstructed line can be drawn have communication potential if in range. A graph represents the current communication-network topology, where vertices represent vehicles and the edges between the vehicles on the bidirectional road segments represent the connection between them. \(n\) cars are moving with speed \(\vec{v}\) in the \((x, y)\) coordinate system. Initially, their positions are assumed to be independent random variables that are uniformly distributed over the range \([X_{\text{min}}, X_{\text{max}}]\) and \([Y_{\text{min}}, Y_{\text{max}}]\). If a path from a starting point to a goal can be extracted from the graph, then a routing capability exists through the network. For our model we determine the future Meeting Point (MP) in advance, at which both vehicles share a road segment and hence a LOS.

3 TASR Protocol

TASR is based on a heuristics for selecting an appropriate path from source to destination. The objective is to select the next forwarding path which gives us the smallest packet delivery delay and maximum probability of packet delivery. Suppose that a situation in our VANET can be modeled as an undirected connectivity graph \(G(V, E)\) using a similar representation as described in [3], where \(V\) is a finite set of cars, and \((i, j) \in E\) represents a wireless link between node \(i\) and node \(j\). A time-dependent function is used to indicate the nodes’ mobility conditions in the network i.e. \(F(t) = f[x(i, t), y(i, t), z(i, t), v(i, t), \theta(i, t), R_i]\), where, \(x(i, t), y(i, t)\) and \(z(i, t)\) represents the position, \(v(i, t)\) the speed, \((i, t)\) the movement direction of the node \(i\) on the road at time \(t\), and \(R_i\) is the unobstructed communication range of node \(i\). TASR bases its routing decisions on capturing real-time traffic density estimation of the next-to-next segments chosen by the last node on the segment, i.e., on local density information only. In Fig1, suppose that source node \(S\) is willing to send a data packet to the destination node \(D\) and has not yet established a communication route. To forward the data packet, \(S\) received the real-time information about the traffic on different segments in the vicinity. There are three possibilities

---

\(^{2}\)Further information is provided online, http://www.mapmechanics.com.
to choose a route, namely $sBEFd(s \rightarrow I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow I_6 \rightarrow d)$, $sBDd(s \rightarrow I_1 \rightarrow I_2 \rightarrow I_5 \rightarrow d)$ and $sACd(s \rightarrow I_1 \rightarrow I_4 \rightarrow I_5 \rightarrow d)$. The traffic densities on the different segments are $A = 2$, $B = 5$, $C = 3$, $D = 0$, $E = 5$, and $F = 3$ cars per length segment where the segment length is longer than the communication range. In our proposed scheme the priority is given to choose that path whose latency is minimum and connectivity between the node is sufficient. The proposed scheme therefore selects $sACd$ deterministically as this route provides the shortest possible path with minimum delay to the expected meeting point (MP), trying to avoid waiting-time overhead and to reduce expected delay. Thus, our TASR protocol follows the following basic principles. (1) Transfer the data packet through wireless channel as far as possible in a single hop, exploiting the LOS. (2) If the packet is to be relayed via a hop into an adjacent segment, the segment with sufficient density in the best next-to-next segment should be chosen. (3) Due to the unpredictable and highly dynamic nature of the VANET, we cannot guarantee that the data packet would be successfully routed along the path.

4 Segment Selection and Data Forwarding

In TASR, segments through which a packet must pass to reach its destination are chosen dynamically and one by one in order to dynamically adapt to the real-time variation in vehicle densities. To select a robust route among the temporarily possible routes, the selecting node in TASR will select a next segment that has a sufficiently connected next-to-next road segment. The node which comes last on the current segment is responsible to select the next-to-next segment which is most attractive based on the current traffic density and the connection probability between the adjacent segments. Hello beacons are used by idle nodes to advertise their presence. Let suppose that the frequency of hello beacons is related to the minimum contact duration. If the frequency of transmission is high, the time necessary for the information to reach the outer bounds of the geographic area is lower. After selection of the next segment as discussed in the above section, data packets forwarding takes place. The sender node has the information about the estimated future position of the destination node as well as the meeting point (MP). After neighbor discovery, the second phase of TASR deals with the individual node determining the next hop for a particular transmission. A node receiving an RREQ packet sends RREP packets back to the source if it has a route to the destination or is the destination itself; otherwise, it will rebroadcast the RREQ. Data packets are sent to the destination after the source node receives the RREP. The routing information is updated to ensure that the best route is chosen along the selected segment. If a link breaks while the route is active, a route error (RERR) message is sent to the source node. The source node may then re-initiate a route discovery process.
5 Calculating ECD

In this section, we study the connectivity of nodes on the bi-directional road segment using the metric \( \text{expected connectivity degree (ECD)} \). The expected connectivity degree or probability is measured in terms of road connectivity along the path between the source and destination. Two vehicles are considered to be connected if their distance is less than the vehicle transmitting range \( R \). A road segment is represented by a tuple of two end points as presented in [3], i.e. \( \text{seg} = (s, d^s), (s, d^d) \), where \( (s, d^s) \) shows the street crossing on one end of the road segment and \( (s, d^d) \) is the next street crossing on the other end so all the segments in a grid is represented as
\[
N = \{(s_1, d_1^s), (s_1, d_1^d) \ldots (s_n, d_n^s), (s_n, d_n^d)\}.
\]

\( ECD = 1 - \sum_{j=1}^{k} (-1)^{j-1} \binom{N_{nodes} - 1}{j} \left(1 - j \frac{R}{L_{seg}}\right) \) where \( k = \min \{N_{nodes} - 1, \left\lceil \frac{L_{seg}}{R} \right\rceil \} \), \( R \) is the transmission range of node, \( L_{seg} \) is the length of each segment and \( N_{nodes} \) is the average number of vehicles on the road segment. Through this formula we can calculate the probability in the form of expected connectivity degree for every neighbor next-to-next segments and after computing we can deterministically choose the best option as the next-to-next segment.

6 Performance Evaluation

The proposed protocol has been implemented in the OMNeT++ 4.5 network simulator. We also used the Simulator for Urban MObiity 0.21.0 (SUMO) to build the simulation scenario and reflect vehicle mobility. Furthermore, the experiments rely on the network framework Veins 3.0, which is well accepted by the research community and certainly improves the reliability of our results, since Veins implements the IEEE 802.11p protocol stack among other features used in our analysis. The test area is a 2000 m \( \times \) 2000 m area within the relatively dense road network of the city of Oldenburg, yielding the grid layout shown in Fig2a. When the destination is reached, another destination is randomly selected. This procedure is repeated until the end of the simulation. We select a small sub-grid as shown in Fig2b that helps us to evaluate the performance and fix the nodes speed to 40km/h. The total number of nodes varies from 60 to 150. IEEE 802.11 DCF protocol is used as the MAC layer transmission protocol. We compared TASR with the established JBR and MURU routing protocols [4, 5]. Simulation results are shown in Fig3. The simulations 3a demonstrate that TASR consistently improves over JBR and MURU concerning packet delivery ratio often by a substantial margin of approximately 50%. From 3b, we note that when the network density is low, 60 to 100, MURU and TASR give constant result, but when the density increase, the overhead of MURU increase dramatically. This is due to the suitable segment selection of TASR which avoid the congested segment and thus a considerable difference can be seen compared with JBR and MURU. 3c shows the average end-to-end delay. In this result TASR performing much better than JBR. However MURU is also slightly increase when the density increase.

7 Conclusion

We proposed a routing protocol called TASR for urban area vehicular networks. TASR is designed to find robust path to delivery data with high delivery ratio, and low overhead and minimum packet delay. A new metric ECD is introduced to compute the most robust segment. Simulation results shows that TASR is performing well over comparative protocol. Related to
Figure 2: (a) City of Oldenburg as a street graph in SUMO (b) A small fragment from Fig2a

Figure 3: (a) PDR vs. density (b) Routing overhead vs. density (c) End-to-end delay vs. density

In this work, more simulation works will be done for better performance evaluation under diverse scenarios. Also, new cross layered architecture will be concerned for further works.

References


A Hierarchical Sparsification Technique for Faster Algorithms in Graphs and Game Graphs

Veronika Loitzenbauer (vl@cs.univie.ac.at)

University of Vienna, Faculty of Computer Science, Austria

We present a sparsification technique, called hierarchical graph decomposition, and survey its recent applications to problems relevant in computer-aided verification. We provide some intuition for which kind of problems this technique can lead to algorithms with improved asymptotic runtimes, in particular for dense graphs. Apart from classical graphs, we also consider Markov decision processes (MDPs) and (two-player) game graphs. For the definitions of the listed problems as well as their application in computer-aided verification we refer the reader to the cited literature. For example, parity games with three priorities can be used to analyze timed automaton games (a model for real-time systems) with reachability and safety objectives [9, 8, 4, 7].

Related Work. Henzinger et al. [10] introduced the hierarchical graph decomposition as a graph sparsification technique to replace ‘m’, the number of edges, with ‘n’, the number of vertices, in the running time bound. They considered the problem of quickly identifying a new connected component in an undirected graph after a batch of edge deletions, motivated by a problem in computational biology. Chatterjee and Henzinger [2] extended the technique to directed graphs, MDPs, and game graphs. They applied it to two problems where the basic algorithms are based on repeated vertex deletions: Computing the maximal end-component decomposition of MDPs (MEC) and computing the winning sets of both players in Büchi games (Büchi).

<table>
<thead>
<tr>
<th>problem</th>
<th>previous runtime</th>
<th>when $m = \Theta(n^2)$</th>
<th>new algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEC</td>
<td>$O(\min{mn^{2/3}, m^{3/2}})$ [1]</td>
<td>$O(n^{8/3})$</td>
<td>$O(n^2)$ [2]</td>
</tr>
<tr>
<td>Büchi (ignoring log factors)</td>
<td>$O(mn)$ [6, 5]</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$ [2]</td>
</tr>
<tr>
<td>Parity-3</td>
<td>$O(mn)$ [14]</td>
<td>$O(n^{3})$</td>
<td>$O(n^{5/2})$ [3]</td>
</tr>
<tr>
<td>Streett (simplified runtime)</td>
<td>$O(\min{mn, m^{3/2}})$ [12]</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$ [3]</td>
</tr>
<tr>
<td>2SCC</td>
<td>$O(mn)$ [15, 13]</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$ [11]</td>
</tr>
</tbody>
</table>

Results. In recent work [3] we applied the technique in two ways: We showed how the runtime analysis can be modified in the case where vertices are not removed from the
graph, which yields a faster algorithm for the nonemptiness problem of Streett automata (STREETT) in dense graphs; and we showed how the technique can be used to combine and speed up known algorithms for the winning sets in Parity games with three priorities (PARITY-3). The latter approach extends to Parity games with an arbitrary number of priorities and is different from the algorithm for Büchi games (which are equal to Parity games with two priorities). For completeness we also list our recent improved algorithm for a classical graph problem (2SCC) \cite{choudhary2013}. We summarize the running time improvements in our and related work in the table above. Note that while for some problems the runtime can only be improved for dense graphs, for other problems the runtime is improved for all cases except when the number of edges \( m \) is in the order of the number of vertices \( n \).

**Decomposition.** We decompose a directed graph \( G = (V, E) \) with \( m = |E| \) edges and \( n = |V| \) vertices into a hierarchy of graphs \( G_i = (V, E_i) \) for levels \( i \in \{1, \ldots, \lceil \log n \rceil \} \). We will define the set of edges \( E_i \) such that \( |E_i| \) is \( O(n \cdot 2^i) \), \( E_{i-1} \subseteq E_i \) for all \( i > 1 \), and \( E_{\lceil \log n \rceil} = E \). Whether an edge \((u, v) \in E\) is included in \( E_i \) will depend on the in- and out-degree of \( u \) and \( v \), and can for game graphs additionally depend on the player to which the vertices on the other end of the incoming or outgoing edges of \( u \) and \( v \) belong. For example, in the algorithms for MEC and STREETT the set \( E_i \) contains all edges \((u, v) \in E\) for which the out-degree of \( u \) is at most \( 2^i \). In the algorithm for PARITY-3 the set \( E_i \) additionally contains the first \( 2^i \) incoming edges of each vertex, where the incoming edges are sorted such that those from Player 2 come first. For now we call all vertices that are not missing outgoing edges in \( G_i \) white.

**Size-Degree Relation.** To see why this definition can be useful, think about a strongly connected component that has no outgoing edges, called a bottom SCC. Let \( C \) be a bottom SCC with \( 2^i \) vertices. Then each vertex in \( C \) must have an out-degree of less than \( 2^i \), i.e., in \( G_i \) all vertices in \( C \) are white. This implies that if we search for bottom SCCs in \( G_i \) that only contain white vertices, then we will detect \( C \). In the algorithms for MEC and STREETT we repeatedly search for such bottom SCCs after some vertices were removed from the graph. This search is started at level \( i = 1 \), and the level is increased by one until the search is successful. Whenever we have to go up to level \( i^* \) to identify such a bottom SCC \( C \), then \( C \) has to contain more than \( 2^{i^* - 1} \) vertices since otherwise it would have been identified at level \( i^* - 1 \). The hierarchical graph decomposition can only be applied if one can show a similar relation between the size of searched sets and the degree of the vertices in this set. For Parity games the searched sets of vertices are part of the winning set of one of the players.

**Runtime Analysis.** A bottom SCC in \( G_i \) can be found in time \( O(|E_i|) \), which is \( O(n \cdot 2^i) \). In the algorithm for MEC the vertices in the identified bottom SCC are then removed from the graph and thus can be charged the work to identify them, which leads to a total runtime of \( O(n^2) \). Although for STREETT the identified bottom SCC is not removed from the graph, we can show the same runtime (for this part of the algorithm) using a parallel search in the reverse graph. For PARITY-3 the time to identify a part of the winning set is proportional to \( |E_i| \) times the size of this part; the runtime bound of \( O(n^{2.5}) \) comes from searching only for parts with at most \( \sqrt{n} \) vertices with the hierarchical graph decomposition and using an \( O(n^2) \) algorithm for larger parts.

**When can it be applied?** All the mentioned problems can be seen as vertex or edge partitioning problems (e.g. partition of vertices into winning set of Player 1 and (parts of) winning set of Player 2). Further, they all have a basic algorithm that iteratively
refines a maintained partition, i.e., there exists a “fast” way to either further divide a
set in the maintained partition or to decide that no further refinement is needed. When,
e.g., the algorithm for Streett identifies a bottom scc, then it recurses on each of the
bottom scc and the subgraph induced by the remaining vertices and thereby refines the
(implicitly) maintained partition. The crucial step in applying the presented technique is
to show a similar size-degree relation, as described above for bottom SCCs, for a set of
vertices or edges that can be used to refine the maintained partition and that can be found
“fast” (e.g. in linear time). We believe that more problems with such a structure exist.

References

[1] K. Chatterjee and M. Henzinger. Faster and Dynamic Algorithms For Maximal End-
Component Decomposition And Related Graph Problems In Probabilistic Verification. In
SODA, pages 1318–1336, 2011.
announced at SODA’11 and SODA’12.
[3] K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved Algorithms for One-Pair and
Design and Verification (GDV), 2006.
[10] M. Henzinger, V. King, and T. Warnow. Constructing a Tree from Homeomorphic Subtrees,
2000.
IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer
1 Introduction

Verification by means of a reachability analysis is based on abstract domains that over-approximate the possible concrete states that a program can reach. The forte of abstract domains is their ability to synthesize new invariants that are not present in the program. However, their inherent approximation may mean that the invariant required to verify a program cannot be deduced. On the contrary, the strength of predicate abstraction used in software model checking is that predicates precisely partition the state space of a program. The challenge here is to synthesize new predicates that eventually suffice to verify a program. This work combines the benefits of both approaches: we synthesize new predicates by observing the precision loss in numeric domains and refine the precision of the numeric domains using the predicates. Our technique is particularly useful for expressing non-convex invariants that are commonly lost when using off-the-shelf numeric abstract domains that are based on convex approximations.

2 Related Work

A common approach to enriching numeric abstract domains to allow expressing non-convex states is to use disjunctive completion, that is, a set of states. In particular, several works have proposed to some variant of a binary decision-diagram (BDD) where decision nodes are labeled with predicates and the leaves are abstract domains [3, 5]. A similar effect is obtained by duplicating the control flow graph (CFG) for each subset of satisfied predicates [6]. In both settings, the number of numeric domains that are tracked may be exponential in the number of predicates. Our work improves over this setup by combining classic predicate abstraction [1] with a single numeric domain, thereby avoiding this exponential duplication of the numeric state. In particular, we present a generic combinator domain that is parameterized over any numeric abstract domain and allows any predicate expressible by the abstract domain. We thereby also generalize over bespoke domains that explicitly track specific disjunctive information, such as disequalities [7].

3 The Predicate Abstract Domain

We present our predicate domain as a co-fibered domain [9], that is, as a domain that is parameterized by another domain. A co-fibered domain $\mathcal{D}$ is parameterized by a child...
domain $C$ that it controls. Their combination is written as $D \triangleright C$ and a state as a tuple $(d,c) \in D \triangleright C$. A transfer function on $D \triangleright C$ may apply any number of transfer functions on its child $c \in C$ before returning a result. Co-fibered domains may be nested. For instance, we combine the predicate domain $P$ with a co-fibered affine equality domain $A$ [8] and a plain interval domain $I$, yielding a stack of domains $P \triangleright A \triangleright I$ where a state $(\bar{\iota}, (\sigma, \iota))$ contains the individual domain states $\bar{\iota} \in P$, $\sigma \in A$ and $\iota \in I$. The predicate domain is given by the lattice $(P \triangleright C, \sqsubseteq_P, \sqcup_P, \sqcap_P)$ where the universe $P : \wp(Pred \times Pred)$ is a finite set of implications $p_1 \rightarrow p_2$ over predicates $p_i \in \wp(Pred)$. Predicates relate linear expressions over the program variables $X$ using a comparison operator $\bowtie$. Note that the set of operators is closed under negation so that the universe of predicates is closed under negation. The choice of implications between only two predicates allows for a simple yet effective propagation of information, as detailed in the next section.

## 4 Transfer Functions and Reductions

The transfer functions of the combined domain state $(\bar{\iota}, c) \in P \triangleright C$ are presented next. In general, a transfer function $\llbracket \delta \rrbracket^P(\bar{\iota}, c)$ applies the corresponding transfer function on the child domain $c \in C$, yielding $(\bar{\iota}', \llbracket \delta \rrbracket^C c)$ where $\bar{\iota}'$ is the new state of the predicate domain. We distinguish three forms of assignments. The first, $\llbracket x = a \bowtie b \rrbracket^P$, assigns the result of a comparison to a variable $x$. Here, the predicate domain removes any predicate that mentions $x$ and adds new predicates based on the comparison. We assume that $x$ is set to one if test $a \bowtie b$ holds and to zero otherwise. Thus, the predicates $x = 0$ and $x = 1$ are used to encode the value of $x$ in the implications. Specifically, the two outcomes $x = 1 \leftrightarrow a \bowtie b$ and $x = 0 \leftrightarrow a \not\bowtie b$ are stored using four implications.

The transfer function $\llbracket x = \text{NonLin} \rrbracket^P$ for non-linear assignment removes all implications in the predicate domain containing $x$. An assignment $\llbracket x = \text{Lin} \rrbracket^P$ of a linear expression to $x$ tries to transform implications containing $x$ if $\text{Lin}$ contains $x$, e.g., $x = x + 1$. For example, consider the predicates state $\bar{\iota} = \{ f = 0 \rightarrow x \leq 5, x \not\leq 10 \rightarrow y = 10 \}$ and the assignment $x = x + 1$ mentioned above. Given the substitution $\sigma = [x/x + 1]$ that describes the change of the state space, we compute $\sigma^{-1} = [x/x - 1]$ that describes how predicates can be transformed so that they are valid in the new state. In the example, applying $\sigma^{-1}$ to the implications yields $\bar{\iota}' = \{ f = 0 \rightarrow x \leq 6, x \not\leq 11 \rightarrow y = 10 \}$.

We now consider the transfer function for an assumption $[a \bowtie b]^P$. The information from the test $a \bowtie b$ is used by the predicate domain to gather further facts about the state. The process of applying these facts to the child domain is called reduction. The reduction is performed as a fixpoint computation and can be seen as an instance of Granger’s framework for reduction by local iteration [2]. Specifically, a function named $\text{fixapply}$ gathers a set of predicates $\bar{p}$ deduced from $a \bowtie b$ and recursively applies each to the child state $c'$, yielding $\llbracket \delta \rrbracket^C c'$

## 5 Lattice Operations and Predicate Synthesis

We continue by detailing the entailment test $(\bar{i}_1, c_1) \sqsubseteq_P (\bar{i}_2, c_2)$. It performs the entailment test $c_1 \sqsubseteq c_2$ on the child domain and tests if all the implications in the right argument $\bar{i}_2$ are entailed by the left argument by calling the function $\text{entailed}(\bar{\iota}', \bar{\iota}, c)$. The latter
Synthesizing Predicates from Abstract Domain Losses

<table>
<thead>
<tr>
<th>$x \in [0, 5]$</th>
<th>$y \in [-5, -1]$</th>
<th>${5 &lt; x \rightarrow 2 \leq y, -1 &lt; y \rightarrow 10 \leq x}$</th>
<th>$c_1 \sqcup_I c_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$[0, 5]$</td>
<td>$[10, 15]$</td>
<td>$[0, 15]$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$[-5, -1]$</td>
<td>$[2, 3]$</td>
<td>$[-5, 3]$</td>
</tr>
</tbody>
</table>

Figure 1: The join of two states in the intervals domain $I$ and the synthesized implications correlating the bounds lost due to the convex approximation.

The function $\text{synth}^I(c_1, c_2)$ returns an implication $p' \rightarrow q' \in I'$ if it is either syntactically entailed in $\bar{i}$ or semantically entailed in the state $c$. Note that neither the syntactic nor the semantic entailment test subsumes the other as both approximate the test differently.

The join $\langle \bar{i}_1, c_1 \rangle \sqcup_P \langle \bar{i}_2, c_2 \rangle$ independently computes a join on the predicate domain and on the child domain. In order to join the implication sets $\bar{i}_1$ and $\bar{i}_2$, we define a function $\text{join}$ that keeps all implications that hold in the respective other state using the $\text{entailed}$ function described above. Note that the semantic entailment test in $\text{entailed}$ is particularly important for the join as one of the predicate domain states may be empty so that the syntactic entailment would discard all implications. The semantic join is able to retain newly inferred predicates in, for example, loop bodies as illustrated later.

In addition to the predicates returned by the $\text{join}$ function, new implications are synthesized from the child domain states using the $\text{synth}^C$ function. The idea is to synthesize implications that characterize the approximation that occurred as part of the $\sqcup_C$ operation. Which synthesized implications are generated depends on the numeric domain. If the predicate language is sufficiently expressive, a domain could potentially characterize all precision losses that occur during a join.

6 Recovering Precision using Relational Information

One strength of our $\text{synth}^I$ function is that it creates relational information, that is, it generates implications between different variables. This relational information enables $\text{fixapply}$ to deduce, from a test of one variable, more precise ranges for other variables. In particular, a test $t$ that separates two states, i.e. $[t]_C^I = c_1$ and $[t]_C^I = \perp$ is enriched by the relational implications so that all losses due to convexity are recovered, that is, $[t]^P(\langle \bar{i}_1, c_1 \rangle \sqcup_P \langle \bar{i}_2, c_2 \rangle) = \langle \bar{i}_1, c_1 \rangle$.

We illustrate this ability using two states $s_1 = \langle \emptyset, \{x \in [0, 5], y \in [-5, -1]\}\rangle$ and $s_2 = \langle \emptyset, \{x \in [10, 15], y \in [2, 3]\}\rangle$. The joined state $s = s_1 \sqcup_P s_2$ is given by $s = \langle\{5 < x \rightarrow 2 \leq y, -1 < y \rightarrow 10 \leq x\}, \{x \in [0, 15], y \in [-5, 3]\}\rangle$. This operation is illustrated in Fig. 1 where the bounds in bold are those that are lost and the arrows indicate which bounds are related by the generated implications. We now show how applying the test $0 < y$ on state $s$ recovers the numeric state in $s_2$ and, analogously, that applying $y \leq 0$ recovers the numeric state of $s_1$. Specifically, when applying the test $0 < y$ on state $s$, the left-hand side of the implication $-1 < y \rightarrow 10 \leq x$ is syntactically entailed, so that $10 \leq x$ is also applied to the child state, yielding the precise value $[10, 15]$ for $x$. The predicate $10 \leq x$ syntactically entails the other implication $5 < x \rightarrow 2 \leq y$. Thus, the predicate $2 \leq y$ is applied to the child state, yielding the precise value $[2, 3]$ for $y$. After that no new predicates are entailed and the recursive predicate application in the function $\text{fixapply}$ stops with the state $s'_2 = \langle\{5 < x \rightarrow 2 \leq y,$
1 \( p = \& \text{some}\_\text{var}; \)
2 \( n = 5; \)
3 \( \text{while} (n \geq 0) \{ \)
4 \( \text{assert}(p \neq 0); \)
5 \( \text{// dereference } p \)
6 \( \cdots \)
7 \( \text{if} (n == 0) \)
8 \( p = 0; \)
9 \( n--; \)
10 \} \)

-1 \( < y \rightarrow 10 \leq x \}, \{ x \in [10, 15], y \in [2, 3] \}. \) Observe that the interval domain is identical to that of \( s_2 \). Analogously, we get a state \( s'_1 \) in which the interval for \( x \) is \([0, 5]\) and for \( y \) is \([-5, -1]\) after applying the opposing condition \( y \leq 0 \).

7 Application to Separation of Loop Iterations

A particularly challenging example from the literature \([4]\) requires that variable values of certain loop iterations are distinguished. The example in Fig. 2 is prototypical for a loop that frees a memory region in its last iteration. The assertion in line 4 expresses that the memory region pointed-to by \( p \) has not yet been deallocated. In order to prove this assertion, an analysis needs to separate the value of the pointer \( p \) in the last loop iteration from its value in all previous iterations. In particular, the example is difficult to prove using convex numeric domains due to a precision loss that occurs when joining the point \( ⟨p, n⟩ = ⟨0, −1⟩ \) at line 10 of the last loop iteration with the states where \( p \neq 0 \) and \( n \geq 0 \).

However, using the simple interval numeric domain and our predicate domain, the example is proved using the fixpoint computation detailed in Fig. 2. In step 1 of the table, \( p \) is initialized to a non-zero address of a variable, which we illustrate by using the value 99. After initializing the loop counter \( n \) in step 2, the loop is entered as the loop condition \( n >= 0 \) is satisfied. In step 5, it is determined that the \( \text{then} \)-branch in line 8 is not reachable. After decrementing \( n \), the state is propagated to the loop head via the back-edge in step 8. At this point, widening is applied. Additional heuristics \([6]\) ensure that the interval \([-1, 5]\) is tried for \( n \), rather than widening \( n \) immediately to \([-\infty, 5]\). By applying the loop condition \( n >= 0 \), a new state for the loop body is obtained in step 9. In step 12, it is observed that the \( \text{then} \)-branch in line 8 is reachable. In the next step in line 9 the states of both branches are joined and the interval domain approximates \( p \) with \([0, 99]\). In the same step, the implications \( 0 < n \rightarrow 99 \leq p, 0 < p \rightarrow 0 \leq n \) are synthesized. In step 14 these predicates are transformed using \( \sigma^{-1} = [n/n + 1] \). This state is joined with the previous state at the loop header at step 15. Our widening heuristic suppresses widening since a new branch in the program has become live \([6]\). Since the resulting numeric state has changed due to the new value of \( p \), the fixpoint computation continues. Note that

<table>
<thead>
<tr>
<th>step</th>
<th>line</th>
<th>intervals</th>
<th>implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>([99, 99])</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>([99, 99])</td>
<td>([5, 5])</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>([99, 99])</td>
<td>([5, 5])</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>([99, 99])</td>
<td>([5, 5])</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>([99, 99])</td>
<td>([5, 5])</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>([99, 99])</td>
<td>([-1, 1])</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>([0, 99])</td>
<td>([5, 5])</td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>([0, 99])</td>
<td>([-1, 5])</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>([0, 99])</td>
<td>([-1, 5])</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>([99, 99])</td>
<td>([0, 5])</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>([99, 99])</td>
<td>([-1, 99] \leq p, 0 &lt; p &lt; -1 \leq n] )</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>([0, 99])</td>
<td>([-1, 5])</td>
</tr>
<tr>
<td>14</td>
<td>18</td>
<td>([0, 99])</td>
<td>([-1, 5])</td>
</tr>
<tr>
<td>15</td>
<td>19</td>
<td>([99, 99])</td>
<td>([-1, 99] \leq p, 0 &lt; p &lt; -1 \leq n] )</td>
</tr>
</tbody>
</table>

Figure 2: Challenging example: freeing a pointer in the last loop iteration.

Frontiers of Formal Methods 2015

180
during the join in step 15, both implications $-1 < n \rightarrow 99 \leq p$, $0 < p \rightarrow -1 \leq n$ are semantically entailed in the current state at the loop head (as computed in step 8’) and therefore kept in the joined state. Evaluating the loop condition in step 16 enforces that $n \geq 0$, that is, $0 \leq n$. The latter predicate syntactically entails the predicate $-1 < n$. Thus, the fixapply function deduces that $99 \leq p$ holds, yielding $p \in [99, 99]$. The assertion holds since intersecting the state at step 16 with $p = 0$ yields ⊥. Thus, at line 4, $p$ cannot be 0 and the assertion holds. Continuing the analysis of the loop observes a fixpoint in step 22. Note that the assertion can also be shown when using standard widening that sets $n$ to $[-\infty, 0]$ in step 8’. However, for the sake of presentation, we illustrated the example with the more precise states.

References


Efficient Shape Analysis of Multithreaded Programs

Suvam Mukherjee (suvam@csa.iisc.ernet.in)
Computer Science and Automation,
Indian Institute of Science, India.

1 Introduction

A large number of commercial programs, like device drivers, manipulate a dynamically allocated memory space (the heap), shared among multiple threads. These threads may concurrently update the heap in different ways. For example, in a device driver, a producer thread may continually add incoming requests to a linked list, while a consumer thread would continually remove requests from the list (assuming it is not empty). Any run of the program would involve intricate interleaved pointer manipulations. An incorrect implementation of such a program could result in the list becoming cyclic, or getting partitioned into different components. In general, it is difficult to prove the correctness of such implementations, owing to the nondeterminism of the scheduler which results in a large number of possible interleavings and heap configurations. Thus, it is necessary to equip the developer with tools, which can algorithmically prove “deep-heap” properties (acyclicity of a list, connectivity of components, etc.)

Coming up with such algorithms is a non-trivial task, as the analysis would have to simultaneously reason about a large number of thread interleavings, along with a large number of possible heap configurations.

Shape analysis for sequential programs is a well researched topic. Some of the prominent works include [1, 2, 5, 9]. The algorithm in [9] uses 3-valued logic to abstract the memory configurations, while the others employ separation logic.

Among the work on shape analysis for multi-threaded programs, in [11], the 3-valued heap abstraction is used to model check concurrent Java programs. This technique may have false negatives, as only a subset of actual execution paths are analyzed. [6] extends the approach in [5] to prove shape properties in certain kinds of concurrent programs. However, the algorithm, and the associated proof of soundness, is complex. [8] provides an elegant way of dealing with multiple threads updating the heap, by decomposing the heap into (possibly overlapping) regions. For precision, each transfer function composes some of these regions, applies the transformer, and decomposes the resulting heap. The technique is very general, and does not rely on any particular heap abstraction. Additionally, the decomposition leads to significant state space savings. However, it suffers from scalability issues, as it explores a superset of actual execution paths.

At a high level, we come up with a novel algorithm which exploits the generality and precision of the shape analysis in [8], while leveraging the efficiency of the concurrent dataflow analysis in [4]. In our algorithm, the user specifies regions of the heap, typically
protected by locks. At each program point, the computation then maps regions to sets of shapes (called region sets). Our analysis runs over sync-CFGs, which only explore a small set of paths. A sync-CFG is a graph, constructed from the control flow graphs of static thread codes, by adding additional edges between specific pairs of synchronization points. A prerequisite for our analysis is that the input concurrent program be free from both data- and region-races. A region \( r \) is relevant at a program point if the statement reads \( r \) along some actual execution. We show that our analysis computes sound and precise shapes for relevant regions.

2 Our Algorithm

We illustrate our algorithm by partially working out an example. Consider the program in Figure 1 (which shows the sync-CFG representation). The program consists of 3 threads, one of which is designated as main. All the variables are assumed to be global. The program maintains two lists, the head of each being pointed to by \( f_1 \) and \( f_2 \) respectively. The main thread performs some initialization, and then spawns a couple of threads. Each spawned thread has a producer component, and a consumer component. Thread 1 nondeterministically adds an entry to the first list, or removes an entry from the second. Thread 2 does the reverse, again in a nondeterministic way.

We seek to verify properties like

1. Do \( f_1 \) and/or \( f_2 \) point to acyclic lists?
2. Are there possible null dereferences in the program?
3. Are there memory leaks in the program?

![Figure 1: Example for our analysis](image-url)
In this example, it is natural to decompose the heap into two regions: one region comprises the memory objects reachable from f1 (region 1), the other comprises the objects reachable from f2 (region 2). We analyze the shapes we obtain after our analysis attains fixpoint. After 22, we might want to ensure that the insert to list 1 does not cause it to become acyclic or disconnected. For this, we might insert assert(acyclic(f1)) immediately before the release. The assert would constitute a read of the first region, and would be non-racy. At fixpoint, the shapes computed for region 1, immediately after 22, are given in Figure 2a. Here, we use 3-valued logic to represent shapes. The obtained shape indicates that after the insert operation, the linear list pointed to by f1 has no cycles or disconnected components (as c is 0 for all the objects, and the reachability predicate rf1 holds for all objects).

Again, if we want to verify whether removing a node from list 2 (in thread 1) maintains the acyclicity and connectivity of the list, we could insert an assert after 32. This would constitute a non-racy read of the region 2. The least fixpoint solution for region 2, after 32, is given in Figure 2b. Both the shapes indicate that, after the deletion, f2 points to an acyclic list with no disconnected components (as c is 0 for all the objects, and the reachability predicate rf2 holds for all objects).

The statement at 57 does a dereference of the object pointed to by f1. If we add an assert(isNotNull(f1)) statement before 57, then the assert passes. This is because the fixpoint solution for region 1, after statement 56, is given in Figure 2c. Clearly, in all the shapes, f1 points to a memory object; thus, a dereference in the next statement cannot cause an error.

3 Future Work

Currently, we are working on the proof of soundness, which provides a theoretical explanation for why the shapes for the relevant regions are soundly approximated. We are also working on a static analysis to detect region races. Finally, we intend to implement our algorithm on the TVLA framework.
References


1 Introduction

In order to analyze the information flow properties of a program, information flow lattices can be used which assign security levels to the program variables [2]. The program is then deemed “secure” if there is no information flow from variables with a higher security level to one with a lower security level. Recent work has shown that results for all possible security lattices can be obtained from a universal information flow lattice [4]. The authors also provide an analysis for this lattice based on type systems in [3]. In some cases, though, their analysis may exhibit unnecessary loss in precision.

Example 1. Consider the following program:

\[
\begin{align*}
y & \leftarrow y + 1; \\
\text{if} \ (secret = 0) \{ x & \leftarrow y; y \leftarrow y + 1; \} \\
\text{else} \{ x & \leftarrow y \}
\end{align*}
\]

This program consists of a branching construct, where the branching condition depends on the secret. Flow-sensitive type systems such as [3] would conclude that the final value of \(x\) may depend on the initial value of secret, since it is assigned to inside a branching construct depending on the value of secret. A more precise analysis, however, may take into account that both branches affect the variable \(x\) in the same way and thus conclude that the value of secret does not influence \(x\). 

2 Weakest Precondition Calculus

Non-interference can be formulated as a hypersafety property [6, 1, 5]. Conceptually, this can be done by comparing all pairs of executions, where the initial values of secret variables are different but public values are equal. Then, non-interference for a variable \(x\) holds if the final values of \(x\) are equal regardless of the initial values of the secret variables. The authors of [5, 7] suggest to use self-compositions of programs to analyze hypersafety properties, which represent all pairs of possible executions of the corresponding program.
Our approach works in two phases: First, a self-composition of the program is constructed using the approach from [5, 7]. Then, we analyze this self-composition using an abstract weakest precondition calculus.

Example 2. Consider again the program from Example 1. One possible self-composition could be:

\[\begin{align*}
    & y \leftarrow y + 1, y \leftarrow y + 1; \\
    & \text{if } (\text{secret} = 0, \text{secret} = 0) \{ \\
    & \quad x \leftarrow y, x \leftarrow y; [y \leftarrow y + 1, y \leftarrow y + 1]; \\
    & \} \text{ else if } (-\text{secret} = 0, \text{secret} = 0) \{ \\
    & \quad x \leftarrow y, x \leftarrow y; [\text{skip}, y \leftarrow y + 1]; \\
    & \} \text{ else if } (\text{secret} = 0, -\text{secret} = 0) \{ \\
    & \quad x \leftarrow y, x \leftarrow y; [y \leftarrow y + 1, \text{skip}]; \\
    & \} \text{ else } \\
    & \quad [x \leftarrow y, x \leftarrow y]; \\
\end{align*}\]

In a self-composition, all operations are pairs of operations of the original program (written in square brackets) with each one operating on its respective copy of the original program state. In our approach, a pair of operations consists either of identical operations, or an operation and a \textit{skip} instruction. A pair of identical operations carry out the same transformation on the pair of states. To achieve a quality self-composition, we try to align similar or identical parts of the program with each other so that equivalent transformations of the state are executed synchronously.

Now, the value of a pair of conditionals may be different on a pair of states. Thus, aligned conditional statements contain all four possible alignments of the two bodies of the original statement. Similarly, aligned iterative statements are split so that it is possible to modify only one of the states, depending on the evaluation of the conditional.

Using the self-composition from Example 2, the reader can easily check that in all branches, \(x\) will be assigned the same value in both copies of the state. Thus, the value of \(x\) at program exit will be equal in both copies regardless of the initial value of \(\text{secret}\). \(\square\)

To analyze these self-compositions, we propose an abstract weakest precondition calculus \(\text{WP}^\#\) which is shown in Figure 1. The result of applying \(\text{WP}^\#\) to a self-composition of a program \(p\) (written \(\text{WP}^\#[p,p](x)\)) is a conjunction of atomic assertions \(y\) which denote “both copies of \(y\) have the same value”. This conjunction is then a necessary precondition for \(x\) to have the same final value in both copies of the state.

Here, \(\text{mod}\) is used to mean the set of variables that are potentially modified (i.e. assigned to) by a statement. \(\text{vars}\) means the set of variables occurring in an expression. We only specify \(\text{WP}^\#\) with a single variable as parameter. However, all right-hand sides distribute over conjunctions and thus \(\text{WP}^\#[p,p](\bigwedge y_i) = \bigwedge \text{WP}^\#[p,p](y_i)\). All preconditions are conjunctions of either atomic assertions, structurally recursive calls, or (in the case of an iterative statement) a monotonic fixpoint iteration. Thus, \(\text{WP}^\#[p,p](x)\) can be computed in polynomial time.

Example 3. Consider \(p\) used in Example 1 with \([p,p]\) from Example 2. Then \(\text{WP}^\#[p,p](x)\) is computed as shown in Figure 2. \(\square\)

We have proven the correctness of \(\text{WP}^\#\) by relating the rules of the calculus to a Hoare calculus on self-compositions.
Theorem 1. Assume that $\text{WP}^# [p, p'] \varphi = \psi$ for program fragments $p$ and $p'$. Then $\psi$ is a precondition for $\varphi$, i.e. whenever $\psi$ holds in the initial states of $p$ and $p'$, $\varphi$ holds in the final states.

Using $\text{WP}^#$, we now have a means to calculate the set of variables on which the final value of a variable $x$ may depend. We can use this method to build an analysis of information flow. As stated earlier, noninterference holds iff the final values of a variable $x$ are always equal regardless of the initial values of secret variables. Thus, we can encode noninterference in the following way: Suppose for a program $p$ that we have a partition of the set of variables into $H$ (the secret variables) and $L$ (the public variables). Then noninterference holds for a variable $x$ iff

$$\bigwedge_{y \in L} y \Rightarrow \text{WP}^# [p, p] (x)$$

Example 4. Continuing Example 3, suppose we have $H = \{\text{secret}\}$ and $L = \{x, y\}$. As we have seen, $\text{WP}^# [p, p] x = y$. Thus, noninterference holds for $x$. \qed

We can also consider a more general formulation of information flow. Here, we do not consider a specific partition of the variables into sets $H$ and $L$, but rather arbitrarily many confidentiality levels (an arbitrary security lattice). As shown in [4], information flow properties for any given security lattice can be proven by analyzing if noninterference holds for all possible partitions of variables into two sets $H$ and $L$. This is called universal information flow analysis.

There are exponentially many possible partitions of the set of variables. However, since $\text{WP}^#$ distributes over conjunctions, we can compute the results $\text{WP}^# [p, p] (x)$ for every variable $x$ in $p$ and combine the results to find results for a specific partition. This means
we only need a polynomial amount of \( \wp^\# \) computations to find results for all possible security lattices. Thus, our approach realizes an analysis of universal information flow that can be computed in polynomial time in the size of the self-composition of the program and its amount of variables.

3 Comparison to Type Systems

We have proven that our analysis is in all cases at least as precise as the analysis detailed in [4, 3]. For that, we have related it to a set-based precondition calculus and proven that our calculus will always yield results that are at least as precise.

4 Conclusion

In this paper, we have introduced an analysis of universal information flow based on an abstract weakest precondition computation of self-compositions of programs. The results can be computed in polynomial time and are always at least as precise as those found by the analysis shown in [4, 3] which is based on type systems. As shown in the running example, there are even simple cases where our analysis finds more precise results.

References

An Analysis of Universal Information Flow based on Self-Compositions


1 Introduction

The growing complexity and dependence on computational systems in our every day life requires formal methods for verification to ensure liveness and safety. Classical model checking analyzes whether a system satisfies certain properties. In the standard approach transitions system are considered as systems. For property specification there exist several logics such as linear temporal logic (LTL) and computation tree logic. A formula in linear temporal logic can be translated to an equivalent nondeterministic Büchi automaton (NBA) which can cause an exponential blowup in the size of the LTL formula. Then the analysis of the system can be carried out in the product of the transition system and the NBA.

Randomized algorithms provide an elegant way for solving problems like leader election, avoiding deadlocks or consensus problems. Analyzing a randomized algorithm can be done via probabilistic model checking (PMC). A basic representation of purely probabilistic behavior is a Markov chain. Adding the concept of nondeterminism yields the notion of a Markov decision process (MDP). As in the classical setting, one can employ LTL for property specification, but for probabilistic systems, the nondeterminism of an equivalent NBA does not allow a direct use. Therefore determinization is necessary. Various determinization procedures exists, Safra’s determinization [9] being the most prominent.

For certain scenarios the circumvention of determinization has been developed. Henzinger and Piterman [4] defined the notion of good-for-games (GFG) restricting the nondeterministic choices. The key idea is, that for every finite prefix \( w \) of an accepted word, one can give an according finite path \( \pi \) in the automaton, such that \( \pi \) can be completed to an accepting infinite path, if \( w \) is completed to an accepting infinite word. Additionally a translation algorithm, called HP-algorithm here, for NBA to GFG parity automata has been proposed, which is amenable to symbolic computation. We will analyze how we can employ GFG automata for probabilistic model checking. On a theoretic level, we will show a GFG-based method to calculate minimal or maximal probabilities for \( \omega \)-regular properties given by a GFG automaton in finite-state Markov decision processes. Afterwards we evaluate the HP-algorithm based on our symbolic implementation and the GFG-based MDP analysis based on a PRISM extension. Details can be found in our main paper [5].
2 GFG based analysis of Markov decision processes

We turn to computing minimal or maximal probabilities in an MDP $\mathcal{M}$ for a given $\omega$-regular language imposed by a nondeterministic $\omega$-automaton $A$. As formal syntax for an MDP we use $\mathcal{M} = (S, Act, P, s_0, AP, \ell)$, where $S$ is the set of states, $Act$ is the set of actions, $P$ is the transition probability function, $s_0$ is the initial state, $AP$ the set of atomic propositions, and $\ell$ the state labeling function. For building a product MDP $\mathcal{M} \otimes A$, we redefine the standard notion (see e.g. [1]) of a product MDP As actions we now take pairs $\langle \alpha, q \rangle$ consisting of an action $\alpha$ of the original MDP and an automaton state $q$. The precise definition for $\mathcal{M} \otimes A$ given a MDP $\mathcal{M} = (S, Act, P, s_0, AP, \ell)$ and a complete nondeterministic automaton $A = (Q, q_0, \Sigma, \delta, \text{Acc})$ is as follows:

$$\mathcal{M} \otimes A = (S \times Q, Act \times Q, P', \langle s_0, q_0 \rangle, AP, \ell')$$

where the transition probability distribution is $P'(\langle s, q \rangle, \langle \alpha, p \rangle, \langle s', q' \rangle) = P(s, \alpha, s')$ if $p = q' \in \delta(q, \ell(s))$ holds, and $P'(\langle s, q \rangle, \langle \alpha, p \rangle, \langle s', q' \rangle) = 0$ otherwise. The labels function is given by the projection to the second component $\ell(\langle s, q \rangle) = q$. Thus, traces of $\mathcal{M} \otimes A$ are infinite words over the alphabet $Q$. Therefore we can treat Acc as property over the paths in $\mathcal{M} \otimes A$.

**Theorem 1 (GFG based analysis of MDP)** For each MDP $\mathcal{M}$ and nondeterministic $\omega$-automaton $A$ as above:

(a) $\Pr_{\mathcal{M} \otimes A}^{\max}(\text{Acc}) \leq \Pr_{\mathcal{M}}^{\max}(A)$

(b) If $A$ is good-for-games then: $\Pr_{\mathcal{M} \otimes A}^{\max}(\text{Acc}) = \Pr_{\mathcal{M}}^{\max}(A)$

From Theorem 1 we know that the methods known from the quantitative analysis of MDPs against deterministic $\omega$-automata specifications are also applicable for GFG automata provided a slight modification of the product MDP.

Kupferman et al. [7, 6] have shown a double exponential blowup as lower bound in the worst case for the size of deterministic automata equivalent to an LTL formula. This proof can be adopted to GFG automata yielding a double exponential lower bound as well.

We implemented three heuristics. The first one concerns the state space. As proposed in [4] we loosened a constraint on the state space. This loosening yields more states but smaller BDD sizes. We refer to this variant as loose variant. Furthermore we implemented the suggested early stop of the construction, if the automaton already satisfies the GFG property. To check whether an automaton is GFG we used an own game-based approach. We refer to this heuristic as iterative variant. At last we used the fact, that the GFG property is preserved by the standard product construction for the union. We refer to this variant as union construction.

3 Implementation and Experiments

We investigate on our implementation of the HP-algorithm LTL2GFG in comparison with LTL2DSTAR. Both implementations start with an LTL formula $\varphi$ and employ LTL2BA to transform $\varphi$ into an (explicitly represented) NBA. According to the HP-algorithm LTL2GFG creates a symbolically represented GFG automaton, whereas LTL2DSTAR creates a deterministic Rabin automaton according to Safra’s algorithm, which is then converted to a symbolic representation.
We report here on formulas known from [3, 10, 2]. All our experiments were carried out on a computer with 2 Intel E5-2680 8-core CPUs at 2.70 GHz with 384 GB of RAM running Linux. We set a memory limit of 10 GB and a time limit of 30 minutes for each formula.

Table 1: Statistics for the automata $A_\phi$ constructed for the 94 benchmark formulas. Number of $A_\phi$ constructed within a given timeframe and a given range of BDD sizes.

<table>
<thead>
<tr>
<th></th>
<th>$A_\phi$ with constr. time</th>
<th>$A_\phi$ with BDD size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>aborted</td>
<td>$&lt;1s$</td>
</tr>
<tr>
<td>LTL2DSTAR std.</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>no opt.</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>LTL2GFG std.</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>std., dynamic</td>
<td>45</td>
<td>34</td>
</tr>
<tr>
<td>loose, dynamic</td>
<td>34</td>
<td>43</td>
</tr>
<tr>
<td>lo., union, dyn.</td>
<td>29</td>
<td>52</td>
</tr>
<tr>
<td>lo., iterative</td>
<td>20</td>
<td>74</td>
</tr>
<tr>
<td>lo., it., union, dyn.</td>
<td>18</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 1 gives an impression how many automata could be constructed within a given time frame. Whereas LTL2DSTAR could generate all automata within 30 minutes, the most of them in a few seconds, LTL2GFG was only able to generate around 60% of the automata with its standard HP algorithm, increasing to 80% for the best variant. The loose variant by itself had a mixed effect, but in conjunction with dynamic reordering was generally beneficial. Our iterative heuristic was successful as well in obtaining smaller automata, because one or two iterations were already sufficient for around 72% of the formulas. For disjunctive formulas the union construction was also very beneficial. Table 2 lists the sizes of the automata in terms of the number of reachable states for some selected formulas. Missing entries means a time-out. The GFG automaton for the formula $\Box\Diamond a \rightarrow \Box\Diamond b$ could not be constructed by the standard HP algorithm. For the loose variant, the automaton has $3.3 \cdot 10^9$ states. With the union heuristics, the size of the automaton shrinks to 5329 states. Table 3 lists the size of the automata in terms of the according BDD size for the transition function.

Both Table 2 and Table 3 were generated with dynamic reordering of the variable order activated.

Table 2: Detailed statistics for example formulas: number of reachable states

| Formula | LTL2GFG standard | loose union, loose it., loose LTL2DSTAR |
|---------|----------------|----------------|----------------|
| true    | 3  | 3  | 3  | 3  | 3  | 2  |
| $a$     | 6  | 8  | 8  | 4  | 4  | 3  |
| $a \rightarrow b$ | 6  | 8  | 50 | 4  | 10 | 3  |
| $\Diamond a$ | 16 | 46 | 46 | 6  | 6  | 2  |
| $\Box\Diamond a$ | 33 | 73 | 73 | 9  | 9  | 2  |
| $\Box\Diamond a \rightarrow \Box\Diamond b$ | 2.2 $\cdot$ 10$^9$ | 3358 | 414 | 4  |
| $a \lor b$ | 16 | 46 | 46 | 6  | 6  | 3  |
| $\Box (a \rightarrow \Diamond b)$ | 33 | 73 | 73 | 9  | 9  | 4  |
Table 3: Detailed statistics for example formulas: size of the transition function BDD

<table>
<thead>
<tr>
<th>Formula</th>
<th>LTL2GFG standard</th>
<th>LTL2GFG loose</th>
<th>LTL2DSTAR union, loose</th>
<th>LTL2DSTAR it., loose</th>
<th>LTL2DSTAR union, it., loose</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>59</td>
<td>46</td>
<td>46</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>a → b</td>
<td>78</td>
<td>48</td>
<td>90</td>
<td>18</td>
<td>32</td>
</tr>
<tr>
<td>♢a</td>
<td>123</td>
<td>85</td>
<td>85</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>□♢a</td>
<td>136</td>
<td>70</td>
<td>70</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>□♢a → □♢b</td>
<td>123</td>
<td>149</td>
<td>149</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>aUb</td>
<td>132</td>
<td>102</td>
<td>102</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>□(a → ♢b)</td>
<td>182</td>
<td>97</td>
<td>97</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

For our PRISM experiment we took a PRISM model [8] from the PRISM benchmark suite. We evaluate the performance using the hand-shaking routine of the or WLAN carrier sense protocoll of IEEE 802.11. We checked six LTL formulas concerning the correct behavior of the Wifi stations even if collisions occurs, that means even if two stations send a message at the same time.

In all experiments the GFG automata generation time was less than a second. This allows a fair comparison of the standard PRISM approach using DRA.

Table 4: Results for model checking with PRISM and different variants of LTL2GFG

<table>
<thead>
<tr>
<th>tGFG &lt; 3 · tSTD</th>
<th>tGFG &lt; 7 · tSTD</th>
<th>tGFG &lt; 20 · tSTD</th>
<th>tGFG ≤ 30 min</th>
<th>Aborted</th>
</tr>
</thead>
<tbody>
<tr>
<td>std. loose</td>
<td>0</td>
<td>5</td>
<td>15</td>
<td>34</td>
</tr>
<tr>
<td>dyn. loose</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>31</td>
</tr>
<tr>
<td>un. loose</td>
<td>0</td>
<td>10</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>it. loose</td>
<td>0</td>
<td>9</td>
<td>21</td>
<td>35</td>
</tr>
<tr>
<td>it., loose, dyn.</td>
<td>0</td>
<td>7</td>
<td>18</td>
<td>35</td>
</tr>
<tr>
<td>it., un. loose</td>
<td>5</td>
<td>11</td>
<td>22</td>
<td>36</td>
</tr>
<tr>
<td>Aborted</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4 gives a summary for the time consumption we measured. We refer to the baseline time spent using the standard approach in PRISM as tSTD and to the time spent using the GFG approach (when there was no timeout) as tGFG. For example, if we consider the loose variant with active iterative approach, in 9 of the 36 cases the running time of PRISM with the GFG approach was within the time spent by the standard PRISM approach multiplied by the factor 7. As can be seen, the variant with loose, union and iterative heuristics fared well in general.

Interestingly, the automata generated with active dynamic reordering in some cases fared significantly worse than those using the initial variable ordering. As PRISM does not support a reordering of the variables, the BDD representation of the GFG automaton is optimized by the dynamic reordering in LTL2GFG for the stand-alone representation. Clearly, this variable ordering may not be optimal for the product with the MDP.

4 Conclusion

We have shown that GFG automata can replace deterministic automata for the quantitative analysis of MDPs against ω-regular specifications without increasing the asymptotic worst-case time complexity. Our experimental results are a bit disappointing, as the generated GFG automata were often larger than DRA generated by the implementation.
of Safra’s algorithm in LTL2DSTAR, both in the number of states and in the symbolic BDD-based representations.

Thus, our empirical results are in contrast to the expectation that the HP-algorithm yields GFG automata that are better suited for symbolic approaches than DRA generated by Safra’s algorithm.

In the context of probabilistic model checking, the GFG-based approach consumed more time and memory than the approach with deterministic automata.

Our negative empirical results might be an artifact of the HP-algorithm, which is – to the best of our knowledge – the only known algorithm for the generation of GFG automata that are not deterministic. Future directions are the design of other algorithms for the construction of succinct GFG automata.

References

When confronted with the question whether a particular software is free of bugs, only a minority of developers would answer “Yes!” This holds even more for high-performance software, where highly complicated algorithms and non-trivial data structures are used.

A measure to counter this problem is to prove the correctness of the particular program\(^1\). There exist projects like seL4 [7] which go that way even for complex systems. But in general this is not feasible for each and every program. Therefore other approaches can be used, which at least increase confidence or ensure some critical properties to hold. An example for such an approach are model checkers. Due to their role as a trust-multiplier their verdict must not be wrong. Now the recursion begins – or as [17] puts it: “Quis custodiet ipsis custodes?” – “Who will watch the watchmen?”

Again, this can be solved in different ways, and Gava et al. [4] give an overview. One particular way has been chosen by us in a previous paper [2]: Let’s verify a model checker! We presented a reference implementation of an LTL model checker for finite-state systems à la SPIN [6], going by the name of CAVA. For its development and verification we use the interactive theorem prover Isabelle/HOL [15]. This allows us to formalize our algorithms and their intended properties in the logic HOL, and also prove that the properties actually hold on those algorithms. Furthermore, Isabelle/HOL has a builtin code-generator [5] which allows exporting our algorithm definitions into a set of target languages (e.g., ML, OCaml, Haskell, Scala) while preserving the correctness properties. Hence, as the final product we receive an executable program.

Our model checker CAVA follows the well-known automata-theoretic approach [20]: Given a finite-state program \( P \) and a formula \( \phi \), two Büchi automata are constructed that recognize the executions of \( P \), and all potential executions of \( P \) that violate \( \phi \), respectively. Then the product of the two automata is computed and tested on-the-fly for emptiness. This thereby touches multiple fields: We have automata theory, temporal logic (LTL), parsers and compilers (for the modeling of \( P \)), and, for the result should be somewhat efficient, efficient data-structures.

Even though we did not reinvent everything from scratch (the incorporation of efficient data structures is, for example, taken care of by the Isabelle Collections Framework [10]), the project spawned multiple results besides the model checker itself: One of particular importance is the development of a refinement framework [12]. This framework allows to specify and reason about an abstract algorithm which can then be refined to more efficient variants while carrying over the properties – the development approach used throughout the whole project. Other results include automata formalizations [8], algorithms for the emptiness check [9], and the translation of LTL formulas into Büchi automata [16].

\(^1\)We are not going to hinder ourselves here by reflecting on how correctness can be actually specified.
My results, which will be described in the following, cover mostly two fields: Easing development and verification of algorithms involving depth-first search [13, 11], and using the same modeling language as SPIN: Promela [14].

As it turned out, most algorithms for on-the-fly emptiness checking are based on depth-first search (DFS) and are very similar to each other (see for example Zhao et al. [22]). In order to reduce the verification work, the goal to develop a general framework for (most) depth-first algorithms evolved. A first approach to this was presented in [13], a second improved version is presented in [11].

The overall idea for such a framework is to distinguish between a general skeleton DFS algorithm (descend through the graph in a, well, depth-first manner), and the specific behavior. The latter is implemented by parametrizing the general algorithm with different hooks specifying the actions to take on different occasions (e.g., reaching a node, finishing a node, aborting condition). To this structure, we add data: The algorithm passes around a state which contains anything deemed useful (set of visited nodes, set of finished nodes, timestamps for both actions, different sets of edges, etc.). Additionally, some opaque extension is added which can be used by the implementation for its own purposes and is the only part of the state which can be changed by the hooks. The final DFS implementation is then represented by its set of hooks.

While such a framework urges a developer to press the algorithm into a corset, it has quite some advantages: A large amount of properties about DFS (e.g., Parenthesis Theorem, White-Path-Theorem) are independent of the specific implementation and thus can be shown beforehand, instead of having to be re-done for each implementation. For simple algorithms, like a cyclicity checker, this results in the complete correctness proof needing only a couple of lines (see [11] for details). Additionally, the structure of the framework allows the author to prove the properties step-by-step. This is an advantage, for one would typically create a large invariant that has to hold throughout the algorithm and would comprise all the different necessary properties. To then prove the validity of this invariant would result in a very large proof, where all the different properties are intertwined.

A further advantage of such a framework is its assistance in generating efficient code for the algorithm. While the support is provided by Isabelle/HOL itself [5], not all definitions are exportable – and even when they are, their performance may be far below par (e.g., when using sets). Using the Isabelle Refinement and Collections Frameworks [12, 10] it was made easier to refine from abstract to concrete level and even automatically replace data structure by more efficient counter-parts, but this still involves some setup and often non-trivial proofs. Especially a recursive algorithm like DFS can be tricky. Using the framework’s inherent separation of concerns is therefore a great help for the developer: For the general skeleton algorithm different optimizations exist (e.g., whether to use a recursive or a tail-recursive style), from which the developer simply has to choose. The effort can thus be focused on the hooks. With [11] we further gained the ability to reduce the content of the DFS state when transferring from the abstract to the concrete level, so that information can be used in the proof which does not need to be collected in the final exported code – this was a great improvement over [13], where each additional field in the state resulted in a performance penalty.

With the help of the framework, we were able to formalize different algorithms involving DFS, from simple cyclicity checkers, over counter-example searches and nested DFS to
complex algorithms like Tarjan’s algorithm for finding the SCCs of a graph [19].

My second larger contribution to the verified model-checker is a formalization of Promela, as described in [14]: Promela [1] is a modeling language, which is mainly used in the model checker SPIN [6]. As SPIN is the explicit state LTL model-checker we compare ourselves with, our aim is to gain as much compatibility to it as possible. This, of course, involves supporting the same input, as otherwise runtimes and even results are not necessarily comparable.

The challenge when adding the support for Promela to CAVA is its lack of proper documentation: While documents exist that detail the semantics of constructs, they are targeted for the modeling audience, i.e., they are quite often rather vague. The existing formal work on Promela (e.g., [21, 3, 18]) is not sufficient, partly because some is outdated, but mainly because there is no guarantee of it being semantically identical to SPIN. Hence, they enhance the understanding, but do not make up for the lack of (official) semantics. For those reasons, some semantic properties had to be determined using SPIN as a black box: Pass the same input to SPIN and CAVA and examine the generated automata. This, of course, still does not guarantee the semantics to match, but, as they generate something executable, can be tested, at least.

Also, as SPIN translates the model to a C program, some semantics are designed to fit C (e.g., `var` and `var[0]` are identical, or the range of types is defined by the C-compiler and not by the language specification) and sometimes hard or impossible to model in a functional language (e.g., `printf`). Furthermore, some semantics in Promela are rather exotic: For example, starting a new process is not a statement but an expression – with the additional restriction that the expression must not be too complex. Because of this, only a subset of Promela could be implemented, but this turned out to be enough to work with 252 of 306 tests of a benchmark suite and it was taken care to bail out on models where the semantics of SPIN and CAVA differ.

Using those benchmarks, we could do the first runtime comparison of CAVA and SPIN on the same models (in [2], the models where given as a Boolean Program for CAVA), showing that CAVA is about 20 times slower than an unoptimized SPIN – a very good result for a verified and automatically generated model checker. One also needs to keep in mind that SPIN generates an explicit program for each model, while CAVA interpretes it at runtime. Hence, one can conclude that we indeed have a very usable result.

References


Strategy Logic: a Powerful Tool for Game Theoretic Issues

Giuseppe Perelli∗(giuseppe.perelli@unina.it)

1 Extended Abstract

In open-system verification, a fundamental area of research is the study of modal logics for strategic reasoning [1]. An important contribution in this field has been the development of Alternating-Time Temporal Logic (ATL⋆, for short), introduced by Alur, Henzinger, and Kupferman [1]. Formally, it is obtained as a generalization of the logic CTL⋆, where the path quantifiers there exists “E” and for all “A” are replaced with strategic modalities of the form “⟨⟨A⟩⟩” and “[A]”, for a set A of agents. These modalities are used to express cooperation and competition among agents in order to achieve a temporal goal. Several decision problems have been investigated about ATL⋆; both its model-checking and satisfiability problems are decidable in 2ExpTime [2], just like they are for CTL⋆.

Despite its powerful expressiveness, ATL⋆ suffers from two strong limitations: 1) strategies are treated only implicitly through modalities that refer to games between competing coalitions and 2) strategic modalities represent coupled ∃∀ and ∀∃ quantifications over strategies. To overcome this problem, Chatterjee, Henzinger, and Piterman introduced Strategy Logic (CHP-SL, for short) [3], which treats strategies in two-player turn-based games as first-order objects. The explicit treatment of strategies makes this logic very useful and more expressive than ATL⋆, however, it still suffers from severe limitations. In particular, it is limited to two-player turn-based games and does not allow different players to share the same strategy, suggesting that strategies have yet to become truly first-class objects in this logic. For example, it is impossible to describe the classic strategy-stealing argument of many real-life combinatorial games.

These considerations has led us to introduce and investigate a new Strategy Logic, denoted SL, as a more general framework than CHP-SL, for explicit reasoning about strategies in multi-agent concurrent games [4]. Syntactically, SL extends the logic LTL by means of strategy quantifiers, the existential ⟨⟨x⟩⟩ and the universal [x], as well as agent binding (a,x), where a is an agent and x a variable. Intuitively, these elements can be read as “there exists a strategy x”, “for all strategies x”, and “bind agent a to the strategy associated with x”, respectively. The price that one has to pay for the expressiveness of SL w.r.t. ATL⋆ is the lack of important model-theoretic properties and an increased complexity of related decision problems. In particular, in [5, 4], it was shown that SL does not have the bounded-tree model property and the satisfiability problem is highly undecidable. Moreover, in [6, 5], it was shown that the model checking problem is

∗Joint works with Orna Kupferman, Fabio Mogavero, Aniello Murano, and Moshe Y. Vardi
nonelementary-complete (we recall that also for CHP-SL it is known to be nonelementary, while it is open the question whether it is nonelementary-hard).

The negative complexity results on the decision problems of SL with respect to ATL$, provide motivations for an investigation of decidable fragments of SL, strictly subsuming ATL$, with a better complexity. In particular, by means of these sublogics, one may understand why SL is computationally more difficult than ATL$. The main fragments we have investigated and studied are Nested-Goal, Boolean-Goal, and One-Goal Strategy Logic, respectively denoted by SL[ng], SL[bg], and SL[1g]. They encompass formulas in a special prenex normal form having nested temporal goals, Boolean combinations of goals, and a single goal at a time, respectively. For goal we mean an SL formula of the type $♭ψ$, where $♭$ is a binding prefix of the form $(α_1,x_1),\ldots,(α_n,x_n)$ containing all the involved agents and $ψ$ is an agent-full formula. In SL[1g], each temporal formula $ψ$ is prefixed by a quantification-binding prefix $℘♭$ that quantifies over a tuple of strategies and binds them to all agents.

As main results about these fragments, we have proved that the satisfiability and model-checking problems for SL[1g] are $2\text{ExpTime}$-complete, thus not harder than the one for ATL$. On the contrary, for SL[ng], the model checking problem is nonelementary and the satisfiability is undecidable. Finally, we observe that SL[bg] includes CHP-SL, the relative model-checking problem relies between $2\text{ExpTime}$ and NonElementary, while the satisfiability problem is undecidable.

To achieve all positive results about SL[1g], we use a fundamental property of the semantics of this logic, called behavioral, which allows us to strongly simplify the reasoning about strategies by reducing it to a step-by-step reasoning about which action to perform. This intrinsic characteristic of SL[1g], which unfortunately is not shared by the other two fragments, asserts that, in a determined history of the play, the value of an existential quantified strategy depends only on the values of strategies, from which the first depends, on the same history. This means that, to choose an existential strategy, we do not need to know the entire structure of universal strategies, as for SL, but only their values on the histories of interest. By means of behavioral, we can solve the SL[1g] decision problems via alternating tree automata in such a way that we avoid the projection operations by using a dedicated automaton that makes an action quantification for each node of the tree model. As this automaton is only exponential in the size of the formula (and independent from its alternation number) and its nonemptiness can be computed in exponential time, we get that both model-checking and satisfiability for SL[1g] are $2\text{ExpTime}$. Clearly, the behavioral property also holds for ATL$, as it is included in SL[1g]. In particular, although it has not been explicitly stated, this property is crucial for most of the results achieved in literature about ATL$ by means of automata.

All the results reported in this paper come from [4, 5, 6, 7]. The interested reader can refer to these works to find more motivations, examples and related material.

1.1 Informal definitions and examples

Due to lack of space, we report here only the informal definitions of the syntax and the semantic framework of Strategy Logic. For a complete and more precise treatment, we strongly recommend to refer to [5].

SL syntactically extends LTL by means of two strategy quantifiers, the existential
\( \langle a, x \rangle \) and the universal \([x]\), and agent binding \((a, x)\), where \(a\) is an agent and \(x\) is a variable. Intuitively, these new elements can be respectively read as “there exists a strategy \(x\)”, “for all strategies \(x\)”, and “bind agent \(a\) to the strategy associated with the variable \(x\)”. SL formulas are built inductively from the sets of atomic propositions \(\text{AP}\), \(V_r\), and agents \(\text{Ag}\), by using the following grammar, where \(p \in \text{AP}\), \(x \in V_r\), and \(a \in \text{Ag}\):

\[
\varphi ::= p | \neg \varphi | \varphi \land \varphi | \varphi \lor \varphi | X \varphi | \varphi U \varphi | \varphi R \varphi | \langle\langle x\rangle\rangle \varphi | \langle\langle \rangle\rangle \varphi | (a, x) \varphi.
\]

SL denotes the infinite set of formulas generated by the above rules.

In order to practice with the syntax of our logic and express game-theoretic concepts through formulas, we describe two examples of important properties that are possible to write in SL. The first concept we introduce is the well-known deterministic Nash equilibrium for Boolean valued payoffs.

**Example 1.1 (Nash equilibrium)** Consider the \(n\) agents \(\alpha_1, \ldots, \alpha_n\) of a game, each of them having, respectively, a possibly different temporal goal described by one of the LTL formulas \(\psi_1, \ldots, \psi_n\). Then, we can express the existence of a strategy profile \((x_1, \ldots, x_n)\) that is a Nash equilibrium (NE, for short) for \(\alpha_1, \ldots, \alpha_n\) w.r.t. \(\psi_1, \ldots, \psi_n\) by using the SL sentence \(\varphi_{\text{NE}} \triangleq \langle\langle x_1\rangle\rangle(\alpha_1, x_1) \cdots \langle\langle x_n\rangle\rangle(\alpha_n, x_n) \psi_{\text{NE}}, \) where \(\psi_{\text{NE}} \triangleq \bigwedge_{i=1}^n (\langle\langle y\rangle\rangle(\alpha_i, y) \psi_i) \rightarrow \psi_i\). Informally, this asserts that every agent \(\alpha_i\) has \(x_i\) as one of the best strategy w.r.t. the goal \(\psi_i\), once all the other strategies of the remaining agents \(\alpha_j\), with \(j \neq i\), have been fixed to \(x_j\). Note that here we are only considering equilibria under deterministic strategies.

**Example 1.2 (Rational Synthesis)** Rational Synthesis \([8, 7]\) is a generalization of the classical problem of synthesis in which the environment is supposed to be composed by a set of agents, each of them having their own temporal goal. For a given solution concept, e.g., dominant strategy, Nash Equilibrium, the system has to satisfy its goal according to the rational behavior of the environment agents. For a given set of temporal goals and a solution concept, there are two variants of rational synthesis. The first one, called cooperative, asks for a strategy profile satisfying the system goal and such that the environment agents are in the solution concept equilibrium. The second, called non-cooperative, asks for a system strategy satisfying the system goal, no matter which is the strategy profile for the environment that is in the solution concept equilibrium. It is easy to see that, for a solution concept \(\gamma\) that is representable in SL by means of \(\Phi^\gamma\), we can formulate the rational synthesis problems as follows:

- **Cooperative**: \(\varphi^C := \langle\langle x_0\rangle\rangle \langle\langle x_1\rangle\rangle \cdots \langle\langle x_k\rangle\rangle (\varphi^\gamma \land \psi_0);\)

- **Non-Cooperative**: \(\varphi^N := \langle\langle x_0\rangle\rangle \langle\langle x_1\rangle\rangle \cdots \langle\langle x_k\rangle\rangle (\varphi^\gamma \rightarrow \psi_0).\)

As semantic framework for our logic language, we use a graph-based model for multi-player games named concurrent game structure \([1]\). Intuitively, this mathematical formalism provides a generalization of Kripke structures and labeled transition systems, modeling multi-agent systems viewed as games, in which players perform concurrent actions chosen strategically as a function on the history of the play.

A concurrent game structure (CGS, for short) is a tuple \(\mathcal{G} \triangleq (\text{AP}, \text{Ag}, \text{Ac}, \text{St}, \text{tr}, \text{ap}, s_0)\), where we respectively have that \(\text{AP}\) and \(\text{Ag}\) are finite non-empty sets of atomic propositions and agents, \(\text{Ac}\) and \(\text{St}\) are enumerable non-empty sets of actions and states,
s₀ ∈ St is a designated initial state, and L : St → 2AP is a labeling function that maps each state to the set of atomic propositions true in that state. Let Dc = Ac × Ag be the set of decisions, i.e., functions from Ag to Ac representing the choices of an action for each agent. Then, tr : St × Dc → St is a transition function mapping a pair of a state and a decision to a state.

A track (resp., path) in a CGS G is a finite (resp., an infinite) sequence of states ρ ∈ St⁺ (resp., π ∈ St⁺) such that, for all i ∈ [0, |ρ| − 1] (resp., i ∈ N), there exists a decision dc ∈ Dc such that (ρ)ᵢ₊₁ = tr((ρ)ᵢ, dc) (resp., (π)ᵢ₊₁ = tr((π)ᵢ, dc)).¹ The set Trk ⊆ St⁺ (resp., Pth ⊆ St⁺) contains all tracks (resp., paths). Moreover, Trk(s) (resp., Pth(s)) indicates the subsets of tracks (resp., paths) starting at a state s ∈ St.

A strategy in a CGS G is a function σ : St → Ac that maps each track to an action. The set of all strategies is denoted by Str.

An assignment in a CGS G is a function χ : Vr ∪ Ag → Str that maps variables in a given set Vr and agents to the set of strategies. The set of assignments over a certain set of variables Vr is denoted by Asg(Vr).

Given a CGS G, for all SL formulas ϕ, states s ∈ St, and assignments χ ∈ Asg(Vr), with Vr be the set of variables occurring in ϕ, the modeling relation G, χ, s |= ϕ is inductively defined as follows.

1. If ϕ is an atomic proposition or its principal scope is a Boolean or temporal operator, the semantic is defined as usual in LTL.
2. For a variable x ∈ Vr and a formula ϕ, it holds that:
   (a) G, χ, s |= ⟨⟨x⟩⟩ϕ if there exists a strategy σ ∈ Str such that G, χₓ→σ, s |= ϕ;
   (b) G, χ, s |= [[x]]ϕ if for all strategies σ ∈ Str it holds that G, χₓ→σ, s |= ϕ.
3. For an agent a ∈ Ag, a variable x ∈ Vr, and a formula ϕ, it holds that G, χ, s |= (a, x)ϕ if G, χₓ→χ(x), s |= ϕ.

Intuitively, at Items 2a and 2b, respectively, we evaluate the existential ⟨⟨x⟩⟩ and universal [[x]] quantifiers over strategies, by associating them to the variable x. Moreover, at Item 3, by means of an agent binding (a, x), we commit the agent a to a strategy associated with the variable x.

Finally, we say that a CGS G is a model of an SL sentence ϕ, in symbols G |= ϕ, if G, χ, s₀ |= ϕ for all assignments χ. An SL sentence ϕ is satisfiable if there is a model for it.

1.2 Work in Progress and Future Directions

In [4], Strategy Logic has been introduced as a new powerful formalism for reasoning about strategies. There, it has been shown that the satisfiability problem is undecidable. In [5], fragments of SL have been introduced and investigated as far as the model-checking problem is concerned. In particular, it turns out that while for SL the model-checking is NonElementary-complete, it is 2ExpTime-complete for SL[1G] (thus not harder

¹The notation (w)ᵢ ∈ Σ indicates the element of index i ∈ [0, |w|] of a non-empty sequence w ∈ Σ⁺.
²By χₓ→σ we are denoting the assignment obtained from χ by substituting only the value of χ(x) with σ.
than that for ATL\(^\star\)). The question about SL\([bg]\) is open. In [6], the satisfiability problem for the fragments we have introduced in [6] has been investigated. It turns out that this problem is undecidable for SL\([bg]\) while it remains 2ExpTime-complete for SL[1g] (as for ATL\(^\star\)).

Out of the above picture, SL[1g] is the biggest known decidable fragment of SL, strictly subsuming ATL\(^\star\). On the other side, the bigger (but undecidable) logic SL\([bg]\) is of major interest. Indeed, it can describe several interesting properties non expressible in SL[1g] such as Nash equilibrium, strong Nash equilibrium, sub-game perfect equilibrium, coalition proof Nash equilibrium, etc. For these reasons, it is our intention to keep investigating SL\([bc]\). It is worth noting that SL\([bc]\) is strictly more expressive than CHP-SL, for which the exact complexity of the model-checking problem is still open. So, solving the model-checking problem for SL\([bc]\) would solve the problem for CHP-SL as well. A possible way to attack the model-checking problem is to proceed by steps, introducing some new fragment of SL\([bg]\) (subsuming SL[1g]) and solving their model-checking problem. In this direction, Mogavero, Murano, and Sauro have defined in [9] a fragment in which it is allowed only the conjunction of goals, while the disjunction is avoided. This logic is called Strategy Logic Conjunctive Goal. They also proved that such a fragment has the behavioral property and then, by applying the procedure explained in [4], its model-checking problem is 2ExpTime-complete.

References

Decomposition of Weighted Timed Automata

Vitaly Perevoshchikov (perev@informatik.uni-leipzig.de)

Institute of Computer Science, Leipzig University, Germany

Timed automata introduced by Alur and Dill [1] are a prominent model for real-time systems. Timed automata form finite representations of infinite-state automata for which various fundamental results from the theory of finite-state automata can be transferred to the timed setting. Although time has a quantitative nature, the questions asked in the theory of timed automata are of a qualitative kind. On the other side, quantitative aspects of systems, e.g., costs, probabilities and energy consumption can be modelled using weighted automata, i.e., classical nondeterministic automata with a transition weight function. The behaviors of weighted automata can be considered as quantitative languages (also known as formal power series) where every word carries a value. Semiring-weighted automata have been extensively studied in the literature (cf. the handbook of weighted automata [6] for surveys).

Weighted extensions of timed automata are of much interest for the real-time community, since weighted timed automata (WTA) can model continuous time-dependent consumption of resources, e.g., memory, power or financial resources. They accept quantitative timed languages where every timed word is mapped to a value, e.g., a real number. In the literature, various models of WTA were considered, e.g., linearly priced timed automata [2], multi-weighted timed automata with knapsack-problem objective [14], and WTA with measures like average, reward-cost ratio [3] and discounting [11]. In [16], WTA over semirings were studied with respect to the classical automata-theoretic questions. However, various models, e.g., WTA with average and discounting measures as well as multi-weighted automata cannot be defined using semirings. For the latter situations, several algorithmic problems were handled. But for several results known from the theories of timed and weighted automata the question whether they also hold for WTA remains open. Moreover, there is no unified framework for WTA.

The main goal of this work is to build a bridge between the theories of WTA and timed automata. First, we develop a general model of timed valuation monoids for WTA. Recall that Nivat’s decomposition theorem [15] is one of the fundamental characterizations of rational transductions and establishes a connection between rational transductions and rational languages. Our first main result is an extension of Nivat’s theorem to WTA over timed valuation monoids. By Nivat’s theorem for semiring-weighted automata described recently in [7], recognizable quantitative languages are exactly those which can be constructed from recognizable languages using operations like morphisms and intersections. The proof of this result requires the fact that finite automata are determinizable. However, timed automata do not enjoy this property. Nevertheless, for idempotent timed valuation monoids which model all mentioned examples of WTA, we do not need determinization.

*Supported by DFG Graduiertenkolleg 1763 (QuantLA)
In this case, our Nivat theorem for WTA is similar to the one for weighted automata. In the non-idempotent case, we give an example showing that this statement does not hold true. But in this case we can establish a connection between recognizable quantitative timed languages and sequentially, deterministically or unambiguously recognizable timed languages.

As a corollary from the established Nivat theorem for WTA, we obtain a connection between recognizable and unambiguously recognizable quantitative timed languages by means of morphism-like mappings. This result could be interesting, since unambiguous automata can have better algorithmic properties than their ambiguous counterparts. For instance, for untimed max-plus automata, the equivalence problem is undecidable [13]. But this problem is decidable for unambiguous max-plus automata [12].

As an application of our Nivat theorem, we provide a characterization of recognizable quantitative timed languages by means of quantitative logics. The classical Büchi-Elgot theorem [4] was extended to both weighted [5, 8] and timed settings [17]. In [16], a semiring-weighted extension of Wilke’s relative distance logic [17] was considered. Here, we develop a different weighted version of relative distance logic based on our notion of timed valuation monoids. In our second main result, we show that this logic and WTA have the same expressive power. For the proof of this result, we use a new proof technique and our Nivat theorem to derive our result from the corresponding result for unweighted logic [17]. Since the proof of our Nivat theorem is constructive, the translation process from weighted relative distance logic to WTA and vice versa is constructive. This leads to decidability results for weighted relative distance logic. In particular, we show the decidability of several weighted extensions of the satisfiability problem for our logic.

Joint work with Manfred Droste [9].

References


Games with Window Quantitative Objectives

Mickael Randour⋆ (mickael.randour@lsv.ens-cachan.fr)

LSV, CNRS & ENS Cachan, France

Abstract based on publications with K. Chatterjee, L. Doyen and J.-F. Raskin [8, 9].

1 Mean-payoff and total-payoff games

Two-player games on graphs. We consider games on finite weighted directed graphs (every edge has an integer weight) with two types of vertices: in player-1 vertices, player 1 chooses the successor vertex from the set of outgoing edges; in player-2 vertices, player 2 does likewise. The game results in an infinite path through the graph, called a play. Players choose their moves according to strategies, mapping from histories to successor states: we consider only deterministic ones (no randomness) and are particularly interested in memory requirements.

Payoffs. The mean-payoff (resp. total-payoff) value of a play is the long-run average (resp. sum) of the edge-weights along the path. Since the limit value does not need to exist for arbitrary plays, two variants are usually considered for each payoff: supremum and infimum variants, respectively considering the sup. and inf. limit values. While traditionally games on graphs with \(\omega\)-regular objectives have been studied for system analysis, research efforts have recently focused on quantitative extensions to model resource constraints of embedded systems, such as power consumption, or buffer size [5]. Quantitative games, such as mean-payoff games, are crucial for the formal analysis of resource-constrained reactive systems, e.g., [2, 20, 3, 4]. For the analysis of systems with multiple resources, multi-dimension games, where edge weights are integer vectors, provide the appropriate framework [7, 22, 11].

Decision problems. The threshold problem for mean-payoff and total-payoff games asks, given a starting vertex, whether player 1 has a strategy that against all strategies of the opponent ensures a play with payoff value at least equal to zero. For both objectives, memoryless winning strategies exist for both players (where a memoryless strategy is independent of the past and depends only on the current state) [12, 14]. This ensures that the decision problems belong to NP \(\cap\) coNP as corresponding one-player games are solvable in polynomial time [16, 23]. Furthermore, they belong to the intriguing class of problems that are in NP \(\cap\) coNP but whether they are in P (deterministic polynomial time) are long-standing open questions. The study of mean-payoff games has also been extended to multiple dimensions\(^1\) where the problem is shown to be coNP-complete [7, 22]. While for one dimension all the results for mean-payoff and total-payoff coincide, our first contribution shows that quite unexpectedly (in contrast to multi-dimensional mean-payoff games) the multi-dimensional total-payoff games are undecidable. We prove it by reduction from the halting problem for two-counter machines [19].

\(^1\)The limit is taken component-wise and compared to a threshold vector.
2 Window objectives

Concept. On the one hand, the complexity of single-dimensional mean-payoff and total-payoff games is a long-standing open problem, and on the other hand, the multi-dimensional problem is undecidable for total-payoff games. We propose alternative objectives named bounded window mean-payoff and fixed window mean-payoff objectives.

In a bounded window mean-payoff objective, instead of the long-run average along the whole play, we consider payoffs over a local bounded window sliding along a play, and the objective is that the average weight must be at least zero over every bounded window from some point on. This objective can be seen as a strengthening of the mean-payoff objective (resp. of the total-payoff objective if we require that the window objective is satisfied from the beginning of the play rather than from some point on, which we call the direct variant), i.e., winning for the bounded window mean-payoff objective implies winning for the mean-payoff objective. In the fixed window objective the window length is fixed and given as a parameter. Winning for the fixed window objective implies winning for the bounded window objective. We depict the operation of the direct fixed window mean-payoff on a hypothetical play in Fig. 1.

Attractive features for window objectives. First, they are a strengthening of the mean-payoff objectives and hence provide conservative approximations. Second, the window variant is very natural. Mean-payoff objectives require the average to satisfy a certain threshold in the long-run (or in the limit of the infinite path), whereas the window objectives provide guarantee on the average, not in the limit, but within a bounded time, and thus provide better time guarantee than the mean-payoff objectives. Third, the window parameter provides flexibility: it can be adjusted specific to applications requirement of strong or weak time guarantee for system behaviors. Finally, we will establish that our variant in the single dimension is more computationally tractable, which makes it an attractive alternative to mean-payoff objectives.

Applicability. In the context of $\omega$-regular objectives, the traditional infinitary notion of liveness has been strengthened to finitary liveness [1], where instead of requiring that good events happen eventually, they are required to happen within a finite time bound. Finitary parity games were studied in [10]. The notion has also been extended to prompt setting where the good events are required to happen as promptly as possible [17]. Our work extends the study of such finite time frames in the setting of quantitative objectives, and our window objectives can be viewed as an extension of finitary conditions for mean-payoff and total-payoff objectives.

Our window variants provide a natural framework to reason about quantitative properties under local finite horizons. Consider a classical example of application with mean-payoff aspects, as presented by Bohy et al. in the context of synthesis from LTL specifications enriched with mean-payoff objectives [3]. Consider the synthesis of a controller for a computer server having to grant requests to different types of clients. The LTL specification can express that all grants should eventually be granted. Adding quantities and a mean-payoff objective helps in defining priorities between requests and associating costs to the delays between requests and grants, depending of the relative priority of the request. Window objectives are usefull for modeling such applications. Indeed, in a desired controller, requests should not be placed on hold for an arbitrary long time. Similarly, if we have two types of requests, with different priorities, and we want to ensure guarantees on the mean waiting time per type of request, it seems natural that an adequate balance between the two types should be observable within reasonable time frames (which can be defined as part of the specification with our new objectives) instead of possible great variations that are allowed by the classical mean-payoff objective.
The maximal window is placed over the initial state.

The window of size 1 is good so the maximal window slides to the next state.

Again, the first window is good and the maximal window slides.

Window size 1 does not suffice: we enlarge the tested window to size 2.

Still a bad window for size 2. But size 3 is still less than $l_{\text{max}} = 4$, so we enlarge again.

Finally, a non-negative mean-payoff is observed. The maximal window can resume sliding.

Figure 1: Illustration of the direct fixed window objective for maximal window size $l_{\text{max}} = 4$ and threshold zero: the maximal window (orange) slides along the play from its starting state and a good window (green) should be found at each step. A window is good if the mean-payoff inside it is non-negative. Bad windows are in red. The tested window is enlarged incrementally up to the maximal window if necessary.

### 3 Main results

The main contributions of our work (along with the undecidability of multi-dimensional total-payoff games) are summarized in Table 1: our results are in bold fonts. For all decidable problems, we also characterize the memory needs.

**Single dimension.** We present a recursive algorithm for the fixed window problem that is polynomial in the size of the graph times the length of the binary encoding of weights times the size of the fixed window. Thus for polynomial windows, we have a polynomial-time algorithm.

For the bounded window, we show that the decision problem is in $\text{NP} \cap \text{coNP}$, and at least as hard as solving mean-payoff games. However, winning for mean-payoff does not imply winning for the bounded window mean-payoff, i.e., the winning sets for mean-payoff games and bounded window mean-payoff games do not coincide. Moreover, winning strategies are also
Table 1: Complexity of deciding the winner and memory required, with \(|S|\) the number of vertices, \(V\) the length of the binary encoding of weights, and \(l_{\text{max}}\) the window size. New results in bold (h. for hard and c. for complete). Infimum (resp. supremum) variants of the mean-payoff (resp. total-payoff) are denoted by \(\text{MP}\), \(\text{MP}^\dagger\), \(\text{TP}\) and \(\overline{\text{TP}}\).

very different, e.g., in mean-payoff games both players have memoryless winning strategies, but in bounded window mean-payoff games we show that player 2 requires infinite memory. This is an interesting reversal of the situation for classical quantitative objectives where player 2 is usually memoryless, even in multi-dimension games or conjunctions with parity (e.g., [11]).

We also show that if player 1 wins the bounded window mean-payoff objective, then a window of size \((|S| - 1) \cdot (|S| \cdot W + 1)\) is sufficient where \(S\) is the state space (the set of vertices of the graph), and \(W\) is the largest absolute weight value. Finally, we show that (i) a winning strategy for the bounded window mean-payoff objective ensures that the mean-payoff is at least zero regardless of the strategy of the opponent, and (ii) a strategy that ensures that the mean-payoff is strictly greater than zero is winning for the bounded window mean-payoff objective.

Multiple dimensions. For the fixed window, we give several results. For arbitrary window sizes, we prove the problem to be EXPTIME-complete. Membership follows from a reduction to exponentially larger co-Büchi games. Hardness is proved in two settings: arbitrary dimensions and weights in \([-1,0,1]\) via a reduction from the membership problem in alternating polynomial-space Turing machines [6], and only two dimensions with arbitrary weights by reduction from countdown games [15]. For polynomial windows, we get PSPACE-hardness via generalized reachability games [13].

Unfortunately, we prove that bounded window games are at least non-primitive-recursive by reduction from the termination problem in reset nets [21, 18]. Decidability remains open.

Wrap-up. The fixed window problem provides an attractive approximation of mean-payoff and total-payoff games that has better algorithmic complexity. In contrast to the long-standing open problem of mean-payoff games, the one-dimension fixed window problem with polynomial window size can be solved in polynomial time; and in contrast to the undecidability of multi-dimensional total-payoff games, the multi-dimension fixed window problem is EXPTIME-complete. In terms of complexity, the problem stands in an interesting middle ground between mean-payoff and total-payoff objectives. For the specific case of polynomial windows, there remains a gap between our exponential-time algorithm and the PSPACE lower bound. Whether we can obtain PSPACE-membership or EXPTIME-hardness for the fixed polynomial window problem in multi-dimension games is an open question. We also established a prohibitive lower bound on the complexity of multi-dimension bounded window games: they are at least non-primitive-recursive-hard. It would still be of theoretical interest to know if those games are decidable or not. Techniques used for the undecidability proof of multi-dimension total-payoff games cannot be extended easily to the bounded window setting.
Games with Window Quantitative Objectives

References


1 From Automatic Structures to Automatic Structures with Parameters...

1.1 Automatic Structures

A structure \((A, R_1, \ldots, R_n)\) is called \emph{automatic presentable} if there is an encoding of its elements as syntactic objects (e.g. finite words, \(\omega\)-words, trees or \(\omega\)-trees) in such a way that the structure’s domain \(A\) and relations \(R_1, \ldots, R_n\) become thereby recognizable by finite (synchronous, multi-tape) automata reading these objects.

Automatic presentations make it possible to handle infinite structures such as infinite graphs or infinite arithmetical structures as finite objects represented by automata in computations. Remarkably, there is an algorithm that, given a first-order logic formula and an automatic presentation, computes an automaton that recognizes the codes of the elements that satisfy the formula. As a consequence the following problems reduce to a membership/emptiness test for the automaton and are decidable for first-order logic (FO) formulae over any automatic structure:

\textbf{Model-Checking:} Given a formula \(\varphi(\overline{x})\) and presentations of \(A\) and a tuple \(\overline{a}\) of elements from \(A\), decide whether \(A \models \varphi(\overline{a})\) holds.

\textbf{Query-Evaluation:} Given a formula \(\varphi(\overline{x})\) and a presentation of \(A\), compute a presentation of \(\varphi^A = \{ \overline{a} \mid A \models \varphi(\overline{a}) \}\)

In particular the first-order theory of any automatic presentable structure is decidable, which originally motivated and continues to motivate a research program into the algorithmic model theory of structures that admit an automatic presentation.

\textbf{Example} Probably the most prominent example of a first-order theory that’s decidable with this method is the Presburger arithmetic, i.e. the first-order theory of the structure \((\mathbb{N}, +)\), which has an automatic presentation as follows: Encode each number \(n\) by its shortest binary representation \(b_0b_1\ldots b_l \in (0 + 1)^*11 + 0\), such that \((b_0b_1\ldots b_l)_2 := \sum_{i \leq l} b_i 2^i = n\). There is a finite automaton \(A_N\) recognizing the domain \((0 + 1)^*11 + 0\).
The $+$-relation on natural numbers $\{(n,m,p) \mid n + m = p\}$ can be recognized by a synchronous 3-tape NFA $A_+$ with two states 0,1 in which it saves the carry of a binary addition it performs with the transition relation $\Delta = \{(i,(a,b,c),j) \mid a+b+i = c+2j\}$. $\delta_{(\mathbb{N},+)} = (A_\mathbb{N},A_+)$ thus constitutes an automatic presentation of the structure $(\mathbb{N},+)$. Automatic presentations were introduced by Khoussainov and Nerode [3] to extend the field of finite model theory to infinite structures in such a way that interesting decision problems remain solvable. Coming from the field of recursive model theory, which is concerned with structures that have recursive predicates (Turing-machine presentable structures), on which FO-model-checking however is not in general decidable, they found with finite automaton presentable structures a good trade-off between algorithmic complexity and structural complexity. In [2] the concept was lifted from automata on finite words to automata that read trees as well as their infinite counter parts. For an overview we refer the reader to [4].

1.2 Automatic Structures with Parameters

1.2.1 Automata with Advice

Recently it was suggested in [1] to extend the research program of automatic structures to structures that are presentable by advice automata. Advice automata are a variant of finite automata. The infinite input tape of an advice automaton contains on each cell an “advice” letter from a finite alphabet $\Gamma$. The input tape is thus initially not filled with blanks, but with an $\omega$-word $\alpha \in \Gamma^\omega$ as a parameter which the automaton reads in parallel with its input word. The languages that are recognizable by an advice automaton can be characterized in terms of monadic-second order logic (MSO) definable languages. According to Büchi’s theorem([5]) $\omega$-regular languages are those, which are MSO-definable in the “blank” $\omega$-word structure $(\mathbb{N}, \text{suc})$. Languages that are recognizable by an $\omega$-automaton with an infinite advice string $\alpha$ are those MSO-definable on the $\omega$-word structure $(\mathbb{N}, \text{suc}, (P_a)_{a \in \Gamma})$, where each $P_a \subseteq \mathbb{N}$ contains the occurrences of the symbol $a \in \Gamma$ in the $\omega$-word $\alpha$. Analogously, following Rabin’s powerful generalization of Büchi’s theorem, the $\omega$-tree regular languages with advice correspond to those $\omega$-tree languages that are MSO-definable on the infinite binary tree with two successor relations expanded by monadic predicates $\{(0,1)^*, \text{suc}_0, \text{suc}_1, (P_a)_{a \in \Gamma}\}$. Like the $\omega$-regular languages, $\omega$-regular languages with advice are thus also effectively closed under first-order operations (union, complement, negation, existential quantification, cylindrification and permutation of variables) and the emptiness problem for an advice automaton can be translated into an MSO-sentence over the advice. Model-checking of FO on an advice automatic presentation therefore remains decidable, if the MSO-theory of the advice $\alpha$ is decidable, or equivalently, if the $\alpha$-acceptance problem for Büchi automata ([7]) is decidable.

1.2.2 $(\mathbb{Q},+)$ and its Peculiarities

The introduction of advice automata into the context of automatic structures was motivated by the long-standing open problem whether the additive group of rationals $(\mathbb{Q},+)$ has an automatic presentation, finally answered in the negative by Tsankov ([6]). What is peculiar about $(\mathbb{Q},+)$ and presumably made the problem difficult to solve is that
(Q, +) is in a sense “almost” automatic presentable. As a subgroup of (R, +) which has an ω-automatic presentation via the infinitary binary representation of reals, the addition of rationals is ω-regular, but the domain becomes non-ω-regular, since in their binary expansion the rationals are represented by the set all ultimately periodic binary sequences, which can easily be seen to be non-ω-regular. The same remains true for other fixed-radix representations (dₐd₋₁...d₀d₋₁d₋₂...b = ∑₁≤i≤l dᵢbᵢ (0 ≤ dᵢ < b) of the reals. Relaxing the requirement of a fixed radix and choosing a mixed-radix representation instead, such as the factorial base representation, it is possible to encode rationals as finite words (d₋₁d₋₁+1...d₀d₁...dₖb = ∑ₘ≤i≤l dᵢbᵢ + ∑₀≤i≤k dᵢ(i + 1)! but this time with digits from an infinite alphabet, i.e. 0 ≤ d_i < i + 2 (i = 0, 1, ..., k) and 0 ≤ d₋₁₋₁ < i + 2 (i = 0, 1, ..., l – 1). Every rational has a representation in this way. More generally we can choose any sequence b := (bᵢ)ᵢ≥₀ of natural numbers with bᵢ ≥ 2 and use this as a mixed-radix base for the representation of rationals (d₋₁d₋₁+1...d₀d₁...dₖb = ∑ₖ≤i≤l dᵢbᵢ with 0 ≤ dᵢ < bᵢ (i = 0, ..., k) and 0 ≤ d₋₁₋₁ < bᵢ (i = 0, ..., l – 1).

The summand of two numbers (d₋₁d₋₁+1...d₀d₁...dₖb + (d₋₁₋₁+1...d₀d₁...dₖ)b = (s₋₁₋₁...sₖ₋₁₋₁) can be performed digit-by-digit with carry in a similar way as for fixed-radix representations.

The carries cᵢ and sums sᵢ are computed according to the following linear recursion.

- c₋₁₋₁ := 0
- c₋₁₋₁+1 := \begin{cases} 1 & \text{if } d₋₁₋₁ + d₋₁₋₁ + c₋₁₋₁ ≥ b₋₁₋₁ \\ 0 & \text{if } d₋₁₋₁ + d₋₁₋₁ + c₋₁₋₁ < b₋₁₋₁ \end{cases}
- s₋₁₋₁ := d₋₁₋₁ + d₋₁₋₁ + c₋₁₋₁ \mod b₋₁₋₁ for \( z = -l₋₁₋₁, ..., k₋₁₋₁, (l₋₁₋₁ := \max\{l₀, l₁\}, k₋₁₋₁ := \max\{k₀, k₁\})

This representation is however still not automatic. An ordinary finite automaton would not be able to recognize when a new digit begins, since the digits can become arbitrarily long and furthermore it would need to know the numbers bᵢ to perform the modulo sum and comparison operations. An automaton with advice however would be able to do it, as was noticed for the factorial base case by [1]. In the general case the advice takes the form bin(b₀)#bin(b₁)#... where bin(bᵢ) is the binary representation of bᵢ and rationals are encoded in the following way

\[
\begin{array}{cccccccc}
\text{bin}(d₀) & \# & \text{bin}(d₁) & \# & \ldots & \# & \text{bin}(dₙ) & \# & \text{bin}(d₁) & \# & \ldots & \# & \text{bin}(dₙ) & \# & \ldots & \# & \text{bin}(dₙ) \\
\text{bin}(d₋₁₋₁) & \# & \text{bin}(d₋₁₋₁) & \# & \ldots & \# & \text{bin}(d₋₁₋₁) & \# & \ldots & \# & \text{bin}(d₋₁₋₁) & \# & \ldots & \# & \text{bin}(d₋₁₋₁) \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\text{bin}(b₀) & \# & \text{bin}(b₁) & \# & \ldots & \# & \text{bin}(b₁) & \# & \text{bin}(b₁) & \# & \ldots & \# & \text{bin}(b₁) \end{array}
\]

In fact for each subgroup of (Q, +) there is an appropriate base such that all and only the elements of this subgroup are representable in this base. The subgroups of (Q, +) correspond up to isomorphism to the torsion-free abelian groups of rank 1. This insight leads to the following theorem.

**Theorem 1.1 ([8]).** Every torsion-free abelian group of rank 1 is advice automatic.
1.2.3 The Limitations of Advice Automatic Structures

Motivated by the positive examples provided by \((\mathbb{Q}, +)\) and its subgroups, we ventured out to seek more examples of structures that might become presentable with advice. Since the territory of automatic structures has already been outlined in quite some detail by other researchers we consulted their map(\cite{9}) with our advices to see whether they could lead us to some hidden treasures. At first however, we have to report a disappointment, though probably a pleasure for those who delight in complicated non-automaticity proofs. The main limitations of automatic structures persist even in the presence of an advice. Thus, most of the non-automaticity results of \cite{9} could be repeated, often requiring different proof methods. The findings are as follows and constitute no deviation of the corresponding classification theorems for adviceless automatic structures:

**Theorem 1.2.**

- The Rado-Graph is not advice automatic. \cite{8}
- Every advice automatic linear order has finite finite-condensation-rank. \cite{8}
- Every advice automatic tree has finite Cantor-Bendixson-rank. \cite{8}
- A countably infinite boolean algebra is advice automatic if and only if it is isomorphic to the interval algebra \(B_{\omega_i}\) of the ordinal \(\omega_i\) for some natural number \(i \geq 1\). (\cite{10})
- There is no countable advice automatic structure that embeds \((\mathbb{N}, \cdot)\). In particular no advice automatic countable torsion-free abelian group of infinite rank.\cite{10}

2 ...to Uniformly Automatic Classes...

An observation that was made when looking at the advice automatic presentation of the subgroups of \((\mathbb{Q}, +)\) is that indeed the same advice automaton suffices for all subgroups, it merely needs different parameters, and furthermore the set of these parameters is an \(\omega\)-regular set itself. In such a situation the FO-theory of the whole class and not merely its individual members is decidable as well, i.e. we can check whether a FO-sentence holds in every member of the class. We call a class of structures that is representable in this manner uniformly automatic. For infinite classes of structures it also makes sense to consider automatic presentations of finite structures and thereby get an automata based decision procedure for their theories. We summarize some of our findings:

**Theorem 2.1** (\cite{10}).

- The class of torsion-free abelian groups of rank 1 is uniformly automatic.
- The class \(VD_2\) of countable scattered linear orders with \(VD\)-rank 2 is uniformly automatic.
- The class of all finite abelian groups is uniformly automatic.
- The class of all abelian groups up to elementary equivalence is uniformly tree-automatic.
- The class of all finite groups which have a direct sum decomposition in which the size of the largest non-abelian factor is bounded by a constant is uniformly automatic.

A further consequence is that FO-model-checking on classes of finite structures that are uniformly automatic (with finite parameters) becomes fixed parameter tractable (FPT),
i.e. when treating the time required to construct the corresponding automaton to a
formula as a constant (needs to be computed only once in practical scenarios anyway),
as well as the time required to compute a parameter of the member of the class (can for
all our classes be done in polynomial time), then FO-model-checking on that class is only
linear in the size of the advice. [[10]]

**Theorem 2.2** (Abu Zaid, [10]). $\text{FO-Model-Checking is FPT on the class of finite abelian groups and the class of finite groups.}$

In [11, Open Question 8.2] the author asks on which classes of finite groups, rings, and
fields FO-model-checking is FPT. To the authors knowledge, our results are the first FPT
results for FO-model-checking on algebraic structures in the literature.

**References**

[1] Alex Kruckman, Sasha Rubin, John Sheridan, and Ben Zax. “A myhill-nerode theorem
for automata with advice.” In GandALF’12, pages 238–246, 2012.


Synthesis is an automated procedure to obtain a correct-by-construction reactive system from a given specification, if one exists. The time complexity for synthesis of a reactive system from a linear temporal logic (LTL) formula is double exponential in the length of the formula [3]. However, limited fragments of LTL and symbolic implementations exhibit more practical time complexities. One such fragment is General Reactivity of rank 1 (GR(1)), where synthesis is possible using a polynomial symbolic algorithm [1].

The availability of efficient synthesis algorithms, as in the case of GR(1), and the guarantee of implementations being correct by construction motivates application in software engineering. One obstacle for engineers is however the difficulty of expressing complete and correct specifications in LTL respectively its limited fragment supported by a synthesis algorithm. For GR(1) this fragment consist of constraints for initial states, safety propositions over the current and successor state, and justice constraints.

We are investigating higher-level and more friendly language constructs extending LTL specifications for synthesis. Our goal is to provide a rich, extensible specification language that benefits from efficient GR(1) synthesis.

**Forklift Specification Example:** Consider the forklift shown Fig. 1. It has a sensor to determine whether it is at a station and two distance sensors to detect obstacles. It has three motors to turn the left and right wheel and to lift the fork. Values read by the sensors are provided as inputs to component `ForkliftController` and its outputs are commands controlling the motors.

An engineer specifies the behavior of the forklift controller to synthesize an implementation. For example, the forklift has to leave the pick-up station between lifting and dropping cargo. The engineer expresses this in Spec. (1).

\[
\text{Occurs } (\neg \text{atStation}) \text{ between } (\text{lift=LIFT}) \text{ and } (\text{lift=DROP}) \quad (1)
\]

**From LTL Patterns to GR(1):** Spec. (1) uses the LTL specification pattern Spec. (2) as identified by Dwyer et al. [2]. Most LTL patterns from this catalog, including the above, are not directly supported by GR(1).

\[
\text{Occurs } p \text{ between } q \text{ and } r := \Box(q \land \neg r \rightarrow (\neg r \lor ((p \land \neg r) \lor \Box \neg r))) \quad (2)
\]

*The presented ideas are joint work with Shahar Maoz.
†J. O. Ringert acknowledges support from a postdoctoral Minerva Fellowship.
Our approach to support this and other patterns is to translate the LTL formula to a deterministic Büchi automaton (DBW), if one exists. We then express (1) the statespace of the DBW using auxiliary variables, (2) the transitions of the initial state as initial assignments, (3) all transitions as safety specifications over current and next state, and (4) the accepting states as justice goals. The resulting translation is in GR(1). A DBW computed for the general LTL expression of Spec. (2) is shown in Fig. 1. Its translation instantiated according to Spec. (1) is shown in List. 1. The pattern describes a safety property (all states of the DBW are accepting) and thus we could omit line 12 in List. 1.

Critical to the usefulness of our approach is that the costly translation of LTL to DBW is done only once for every pattern from the catalog or every new pattern a user might wish to add to her library. This works because patterns are only instantiated with propositions (not with nested temporal operators). In case no DBW exists the pattern cannot be added to the specification language.

We are working on providing an engineering friendly, high-level specification language that benefits from efficient GR(1) synthesis, is extensible, and is correct by construction due to the automated translation sketched above. Early investigation shows that many LTL patterns from [2] can be supported.
References


Inter-procedural Two-Variable Herbrand Equalities are in PTIME

Stefan Schulze Frielinghaus (schulzef@in.tum.de)

PUMA, Technische Universität München, Germany

Inferring invariants in programs which make use of complex operators as e.g. taking the square root or if the meaning of operators is even unknown, obvious methods for instance based on linear algebra fail. However, a true Herbrand equality holds irrespectively of the meaning of the occurring operator symbols. Hence, for a Herbrand equality to hold, the terms on the left- and right-hand side must be syntactically equivalent up to occurring program variables. Such an interpretation then enables us to infer invariants even if complex operators are used.

The problem of inferring Herbrand equalities is known since the 70s where it was introduced by Cocke and Schwartz [1] as the global value numbering problem. Since then algorithms have been developed in order to infer all valid intra-procedural Herbrand equalities. However, in presence of recursive procedures very little is known. Seidl et al. [6] showed that the intra-procedural techniques can be extended to programs with procedures and local but no global variables. Furthermore, the presented techniques are strong enough to infer all valid Herbrand constants for programs with recursive procedures possibly containing local and global variables, i.e., invariants of the form \( x = t \), where \( t \) is ground. Another feasible case of invariants is obtained if only assignments are taken into account where the right-hand side contains at most one program variable. That means an assignment as e.g. \( x := f(5,y) \) is considered while assignments as \( x := f(y,y) \) or even \( x := f(y,z) \) are ruled out. Petter [7] shows that for such programs all inter-procedurally valid Herbrand equalities can be inferred.

For programs only with assignments where the right-hand side contains at most one program variable but possibly multiple times, we showed in a recently accepted paper [9] (joint work with Michael Petter and Helmut Seidl) that all inter-procedurally valid two-variable Herbrand equalities can be effectively inferred (the long version is given in [8]). The methods there are based on monoidal techniques which have been established by Gulwani and Tiwari [3]. They showed that if only assignments using unary operators are taken into account, then all inter-procedurally valid Herbrand equalities can be inferred. The novel result of our paper is that a program variable might occur several times in the right-hand side of an assignment as for example in \( x := f(g(y),5,y) \). That means our approach is not limited to strings but general terms containing at most one variable but possibly multiple times. In order to arrive at an effective algorithm we establish an approximate notion of subsumption and compactness which are strong enough in order to infer all inter-procedurally valid two-variable Herbrand equalities. In our case, approximate means that we may not always detect that an equality is subsumed but we fail only finitely many times.
The analysis is based on procedure summaries representing the weakest pre-conditions of finitely many generic post-conditions which contain template variables. A template variable is nothing else than a second-order variable which gets instantiated by an arbitrary term containing no program variable but a dedicated place holder for the argument of it. That means, in our case we demand that the argument of a second-order variable occurs at least one time in the resulting term. Deciding subsumption between conjunctions of equalities which contain second-order variables is subtly related to second-order unification which is in general undecidable [2] and even undecidable in the case if only one unary second-order variable is considered which was shown by Levy and Veanes [4]. Though the latter result is not immediately applicable to our problem, since we consider only terms with at most one program variable while Levy and Veanes allow arbitrary many variables. Hence, the problem of second-order unification of our restricted case is still open to the best of our knowledge. Yet in order to decide subsumption and arrive at effective representations for all occurring weakest pre-conditions, we showed for almost all values possibly computed at run-time, that they can be uniquely factorized into tree patterns and a terminating ground term. Moreover, we introduced an approximate notion of subsumption which is effectively decidable and ensures that finite conjunctions of equalities may not grow infinitely. Based on these technical results, we realized an effective fixpoint iteration to infer all inter-procedurally valid Herbrand equalities for programs where each right-hand side of an assignment contains at most one program variable but possibly multiple times.

Here we study the complexity of our algorithm and show that it is in PTIME. For that we first note that the terms occurring during the computation might be exponentially large in the size of the program. Hence, we need a compact representation for all occurring terms which supports all basic term operations as e.g. prefix/suffix computation in polynomial time. Ongoing we examine the algorithm with respect to terms which are represented in such a compact form. We will observe that one of the main parts of the subsumption algorithm is a generalization of Euclidians algorithm which has logarithmic complexity in the smaller input. Altogether with our compactness result from previous work we finally show that our algorithm has polynomial time complexity.

1 Two-Variable Herbrand Equalities

In the following we consider only imperative programs with assignments where each right-hand side contains at most one program variable which might occur several times. Furthermore, the program model makes use of non-deterministic branching and recursive functions. In order to infer all invariants for such programs, we summarize the effect of procedures for multiple, but finitely many, generic post-conditions only. A two-variable generic post-condition is of the form $Ax \equiv By$ where $A,B$ are template variables, i.e., variables which range over arbitrary terms not containing program variables but a special place holder (denoted as •) in which the argument gets substituted. Computing weakest pre-conditions of a generic post-condition then operates on equalities of the form $As \equiv Bt$ where $s,t$ are terms possibly containing occurrences of a single program variable. In [9] we show that for finite conjunctions of such equalities there exists an approximate notion of subsumption which is effectively decidable. Furthermore, we show that for each occurring conjunction of a set $E$ of equalities there exists a finite conjunction of a subset in $E$ which
is equivalent. As already mentioned in [9] we show an approximate notion of subsumption and compactness. We obtain these results by limiting the values a template as well as a program variable might receive to a suitable superset of values which might be computed by the program of interest. We denote this superset of possible values by $T$ which can be divided into a finite set of so called small ground values $S$ and a (possibly infinite) set of large ground values. For the set of large ground values we show that each term can be uniquely factorized into tree patterns, while we cannot uniquely factorize small ground values. However, the latter ones are only finitely many. These observations then allow us to apply monoidal techniques from [3] in order to effectively decide subsumption, or more precisely, $T$-subsumption, i.e., subsumption with respect to the set $T$ of possible ground values only.

## 2 Polynomial Time Complexity

Before we examine the decision procedure for $T$-subsumption, we first have a look at a program fragment consisting of procedures $p_n$ and two global program variables $x$ and $y$:

$$
\begin{align*}
    p_i \{ & p_{i-1}(); p_{i-1}(); \} \\
    p_0 \{ & x := f(x,x); y := f(y,y); \}
\end{align*}
$$

The weakest pre-condition of a generic post-condition $Ax = By$ for a procedure $p_n$ is then given by a single equality $Af^{x}(x,x) = Bf^{y}(y,y)$ with exponentially large terms on both sides of the equality. Hence, we immediately note that basic operations as e.g. computing largest common prefix on such terms cannot be done in polynomial time. However we also observe that a program itself is an implicit representation of succinct context-free grammars. Furthermore, each context-free grammar can be transformed into Chomsky normal form in polynomial time. A straight-line program (SLP, for short) is a context-free grammar $G$ in Chomsky normal form where $L(G) = \{ w \}$. Since $L(G) = \{ w \}$, the grammar is non-recursive. Such a grammar then defines the word $w$ by the derivation of the start non-terminal. Hence, each term occurring during a WP computation can be encoded into succinct SLPs which are linear in the size of the program. In [8] we show that the following basic operations on terms which are represented by SLPs can be performed in polynomial time.

**Corollary.** The following term operations are polynomial on SLPs:

1. The balance $|w|$ of a term $w$;
2. The power $w^r$, $r \geq 1$ of a term $w$;
3. The largest common prefix/suffix of two terms;
4. Concatenating two terms;
5. Splitting a term $w$ into two terms $u$ and $v$ such that $w = uv$.

Consider two distinct equalities $As_i \equiv Bt_i$, $i = 1, 2$ such that the conjunction of them is $T$-satisfiable. Assuming that the variables $x, y$ take large values only, we can uniquely factorize the terms. Both equalities then imply the monoidal identity
As_1s_2^{-1}A^{-1} = Bt_1t_2^{-1}B^{-1}. Let the balance |w| of a factorized term w equal the difference of the number of positive and negative letters in w. In [9] we then come up with the following lemma and theorem:

**Lemma 1.** If |u| = |u'| = 0, then the equality AuA^{-1} = Bu'B^{-1} either is trivial, is equivalent to an equality As = B or an equality A = Bs for some s or is contradictory.

**Theorem 2.** Two equalities AuA^{-1} = Bu'B^{-1} and AvA^{-1} = Bv'B^{-1} are effectively equivalent either to one solved equation, or to a single equation or are contradictory.

By that we state our first complexity result:

**Lemma 3.** Deciding subsumption between two equalities AuA^{-1} = Bu'B^{-1} and AvA^{-1} = Bv'B^{-1} takes polynomial time.

**Proof.** W.l.o.g. assume that |u| ≥ |v|, then a third equality AwA^{-1} = Bu'B^{-1} with |w| = |u| mod |v| is derived. If |w| = 0, then according to Lemma 1 we can decide if an equality is subsumed or not. Otherwise, |w| > 0, and a further iteration is performed including the equalities AvA^{-1} = Bv'B^{-1} and AwA^{-1} = Bu'B^{-1} where |v| ≥ |w| holds. This algorithm is a generalization of Euclids algorithm which has logarithmic complexity in the size of the smaller input [5, pp. 21–22].

In each iteration a new term w is constructed such that w = uv^{-r} where r is maximal in |u| ≥ r · |v|. If the term v is represented by a SLP G, then a SLP G' which represents the term v^r can be constructed in logarithmic time with |G'| ≤ |G| + 2 · log_2(r). Since Euclids algorithm performs at most logarithmic many iterations in the size of the smaller input, and in each iteration we increase the grammar by a summand depending on the size of the grammar itself, and according to Corollary 2 all basic term operations can be performed in polynomial time, the assertion of the theorem follows.

We showed that L-subsumption, i.e., subsumption w.r.t. large values only, takes polynomial time for equalities of the form Asx = Bty. In [8] we additionally show that T-subsumption takes polynomial time, too. Furthermore, we show that the same holds for different formats of equalities (e.g. one program variable As = Btx or no program variable at all As = Bt). Hence we come up with the following theorem:

**Theorem 4.** Approximate T-subsumption for finite conjunctions of two-variable equalities is decidable in polynomial time.

In [9] we also showed that each occurring conjunction \( \phi \) during a WP computation is approximately T-subsumed by a conjunction with at most \( O(n^2 \cdot m^2) \) equalities in \( \phi \) where \( n \) is the number of program variables and \( m \) is the cardinality of the set of small terms. Since the set of small terms is part of the input, we then have:

**Corollary.** Assume that all right-hand sides of assignments in an arbitrary program contain at most one variable. Then all inter-procedurally valid two-variable Herbrand equalities can be inferred in polynomial time in the size of the program.
3 Conclusion

Here we showed that deciding $L$-subsumption between equalities of the form $A \equiv B$ takes polynomial time. In [8] we show that subsumption w.r.t. small and large values takes polynomial time, too. Additionally we show that there exist only finitely many formats of equalities which we have to consider and that each conjunction $\phi$ of such equalities is approximately $T$-subsumed by a finite conjunction of at most polynomial many equalities in $\phi$. Hence it immediately follows that the overall analysis is polynomial.

References


On Promptness in Parity Games

Loredana Sorrentino∗(loredana.sorrentino@unina.it)

1 Introduction

In this article we keep working on two-player parity games, under the prompt semantics, over colored (vertexes) arenas with or without weights over edges. In the sequel, we refer to the latter as colored arenas and to the former as weighted arenas. We give a clear picture of all different extended parity conditions introduced in the literature working under the prompt assumption. In particular, we analyze their main intrinsic peculiarities and possibly improve the complexity class results related to the game solutions. Furthermore, we introduce new parity conditions to work on both colored and weighted arenas and study their relation with the known ones. As a main contribution, we introduce and study three new parity conditions named full parity (FP), prompt parity (PP) and full-prompt parity (FPP) condition, respectively. The full parity condition is defined over colored arenas and, in accordance to the full semantics, it simply requires that all requests must be responded. See Table 1 for a schematic view of this argument. We prove that the problem of checking whether player ∃ wins under the full parity condition is in $\text{PTime}$ [3].

The prompt parity condition, which we consider on both colored and weighted arenas, requires that almost all requests are responded within a bounded cost, which we name here delay. The full-prompt parity condition is defined accordingly. Observe that the main difference between the cost parity and the prompt parity conditions is that the former is a conjunction of two properties, in each of which a possibly different set of finite requests can be ignored, while in the latter we indicate only one set of finite requests to be used in two different properties. Nevertheless, since the quantifications of the winning conditions range on co-finite sets, we are able to prove that prompt and cost parity conditions are semantically equivalent. We also prove that the complexity of checking whether player ∃ wins the game under the prompt parity condition is $\text{UPTIME} \cap \text{CoUPTIME}$, in the case of weighted arenas [3]. So, the same result holds for cost parity games and this improves the previously known results. Observe that, on colored arenas, prompt and full-prompt parity conditions correspond to the finitary and bounded-finitary parity conditions [1], respectively. Hence, both the corresponding games can be decided in $\text{PTime}$. We prove that for full-prompt parity games the $\text{PTime}$ complexity holds even in the case the arenas are weighted. Finally, by means of a cubic translation to classic parity games, we prove that bounded-cost parity over weighted arenas is in $\text{UPTIME} \cap \text{CoUPTIME}$ [3], which also improves the previously known result about this condition. All the results reported in this paper come from [3]. The interested reader can refer to this work to find more motivations, examples and related material.

∗Joint work with Fabio Mogavero and Aniello Murano
2 Preliminaries

In this section, we give the concepts of two-player turn-based arena, payoff-arena, and game. Furthermore we introduce some notation to formally define all addressed winning conditions.

An arena is a tuple $A \triangleq (Ps, Ps, Mv)$, where $Ps$ and $Ps$ are the disjoint sets of existential and universal positions and $Mv \subseteq Ps \times Ps$ is the left-total move relation on $Ps \triangleq Ps \cup Ps$. The order of $A$ is the number $|A| \triangleq |Ps|$ of its positions. An arena is finite if it has finite order. A path (resp., history) in $A$ is an infinite (resp., finite non-empty) sequence of vertexes $\pi \in Pth \subseteq Ps$ (resp., $\rho \in Hst \subseteq Ps^+$) compatible with the move relation, i.e., $(\pi_i, \pi_{i+1}) \in Mv$ (resp., $(\rho_i, \rho_{i+1}) \in Mv$), for all $i \in N$ (resp., $i \in [0, |\rho| - 1]$), where $Pth$ (resp., $Hst$) denotes the set of all paths (resp., histories). Intuitively, histories and paths are legal sequences of reachable positions that can be seen, respectively, as partial and complete descriptions of possible outcomes obtainable by following the rules of the game modeled by the arena. An existential (resp., universal) history in $A$ is just a history $\rho \in Hst \subseteq Hst$ (resp., $\rho \in Hst \subseteq Hst$) ending in an existential (resp., universal) position, i.e., $\text{lst}(\rho) \in Ps$ (resp., $\text{lst}(\rho) \in Ps$). An existential (resp., universal) strategy on $A$ is a function $\sigma_\exists \in \text{Str}_\exists \subseteq \text{Str}_\exists \rightarrow Ps$ (resp., $\sigma_\forall \in \text{Str}_\forall \subseteq \text{Str}_\forall \rightarrow Ps$) mapping each existential (resp., universal) history $\rho \in \text{Hst}_\exists$ (resp., $\rho \in \text{Hst}_\forall$) to a position compatible with the move relation, i.e., $(\text{lst}(\rho), \sigma_\exists(\rho)) \in Mv$ (resp., $(\text{lst}(\rho), \sigma_\forall(\rho)) \in Mv$), where $\text{Str}_\exists$ (resp., $\text{Str}_\forall$) denotes the set of all existential (resp., universal) strategies. Intuitively, a strategy is a high-level plan for a player to achieve his own goal, which contains the choice of moves as a function of the histories of the current outcome. A path $\pi \in Pth(v)$ starting at a position $v \in Ps$ is the play in $A$ w.r.t. a pair of strategies $(\sigma_\exists, \sigma_\forall) \in \text{Str}_\exists \times \text{Str}_\forall$ of the form for short $\pi \in N$, it holds that if $\pi_i \in Ps$ then $\pi_i = \sigma_\exists(\pi_i)$ else $\pi_i = \sigma_\forall(\pi_i)$. Intuitively, a play is the unique outcome of the game given by the player strategies. The play function $\text{play} : Ps \times (\text{Str}_\exists \times \text{Str}_\forall) \rightarrow Pth$ returns, for each position $v \in Ps$ and pair of strategies $(\sigma_\exists, \sigma_\forall) \in \text{Str}_\exists \times \text{Str}_\forall$, the $(\sigma_\exists, \sigma_\forall)$-play $\text{play}(v, (\sigma_\exists, \sigma_\forall))$.

A payoff arena is a tuple $\hat{A} \triangleq (A, Pf, pf)$, where $A$ is the underlying arena, $Pf$ is the non-empty set of payoff values, and $pf : Pth \rightarrow Pf$ is the payoff function mapping each path to a value. The order of $\hat{A}$ is the order of its underlying arena $A$. A payoff arena is finite if it has finite order. The overloading of the payoff function $pf$ from the set of paths to the sets of positions and pairs of existential and universal strategies induces the function $pf : Ps \times (\text{Str}_\exists \times \text{Str}_\forall) \rightarrow Pf$ mapping each position $v \in Ps$ and pair of strategies $(\sigma_\exists, \sigma_\forall) \in \text{Str}_\exists \times \text{Str}_\forall$ to the payoff value $pf(v, (\sigma_\exists, \sigma_\forall)) \triangleq pf(\text{play}(v, (\sigma_\exists, \sigma_\forall)))$ of the corresponding $(\sigma_\exists, \sigma_\forall)$-play.

A (extensive-form) game is a tuple $G \triangleq (A, Wn, v_0)$, where $A = (A, Pf, pf)$ is the underlying payoff arena, $Wn \subseteq Pf$ is the winning payoff set, and $v_0 \in Ps$ is the designated initial position. The order of $G$ is the order of its underlying payoff arena $A$. A game is finite if it has finite order. The existential (resp., universal) player $\exists$ (resp., $\forall$) wins the game $G$ iff there exists a $v_0$-total existential (resp., universal) strategy $\sigma_\exists \in \text{Str}_\exists(v_0)$ (resp., $\sigma_\forall \in \text{Str}_\forall(v_0)$) such that, for all $v_0$-total universal (resp., existential) strategies $\sigma_\forall \in \text{Str}_\forall(v_0)$ (resp., $\sigma_\exists \in \text{Str}_\exists(v_0)$), it holds that $pf(v_0, (\sigma_\exists, \sigma_\forall)) \in Wn$ (resp., $pf(v_0, (\sigma_\exists, \sigma_\forall)) \notin Wn$).

Before continuing, we introduce some notation to formally define all winning conditions. A colored arena is a tuple $\tilde{A} \triangleq (A, Cl, cl)$, where $A$ is the underlying arena, $Cl \subseteq N$ is the non-empty sets of colors, and $cl : Ps \rightarrow Cl$ is the coloring function mapping each
position to a color. Similarly, a (colored) weighted arena is a tuple \( \mathcal{A} \triangleq \langle A, Cl, cl, Wg, wg \rangle \), where \( \langle A, Cl, cl \rangle \) is the underlying colored arena, \( Wg \subseteq \mathbb{N} \) is the non-empty sets of weights, and \( wg : Mv \rightarrow Wg \) is the weighting functions mapping each move to a weight. The overloading of the coloring (resp., weighting) function from the set of positions (resp., moves) to the set of paths induces the function \( cl : Pth \rightarrow Cl^\omega \) (resp., \( wg : Pth \rightarrow Wg^\omega \)) mapping each path \( \pi \in Pth \) to the infinite sequence of colors \( cl(\pi) \in Cl^\omega \) (resp. weights \( wg(\pi) \in Wg^\omega \)) such that \( (cl(\pi)_i) = cl(\pi_i) \) (resp., \( (wg(\pi)_i) = wg((\pi_i, \pi_{i+1})) \)), for all \( i \in \mathbb{N} \).

Every colored (resp., weighted) arena \( \mathcal{A} \triangleq \langle A, Cl, cl \rangle \) (resp., \( \mathcal{A} \triangleq \langle A, Cl, cl, Wg, wg \rangle \)) induces a canonical payoff arena \( \hat{\mathcal{A}} \triangleq \langle A, Pf, pf \rangle \), where \( Pf \triangleq Cl^\omega \) (resp., \( Pf \triangleq Cl^\omega \times Wg^\omega \)) and \( pf(\pi) \triangleq cl(\pi) \) (resp., \( pf(\pi) \triangleq (cl(\pi), wg(\pi)) \)).

Along a play, we interpret the occurrence of an odd priority as a “request” and the occurrence of the first bigger even priority at a later position as a “response”. Then, we distinguish between prompt and not-prompt requests. In the not-prompt case, a request is responded independently from the elapsed time between its occurrence and response. Conversely, in the prompt case, the time within a request is responded has an important role. It is for this reason that we consider weighted arenas. So, a delay over a play is the sum of the weights over of all the edges crossed from a request to its response. We now formalize these concepts. Let \( c \in Cl^\omega \) be an infinite sequence of colors. Then, \( Rq(c) \triangleq \{ i \in \mathbb{N} : c_i \equiv 1 \pmod{2} \} \) denotes the set of all requests in \( c \) and \( rs(c, i) \triangleq \min\{ j \in \mathbb{N} \mid i \leq j \land c_i \leq c_j \land c_j \equiv 0 \pmod{2} \} \) represents the response to the requests \( i \in Rs \), where by convention we set \( \min \emptyset \triangleq \omega \). Moreover, \( Rs(c) \triangleq \{ i \in Rq(c) : rs(c, i) < \omega \} \) denotes the subset of all requests for which a response is provided. Now, let \( w \in Wg^\omega \) be an infinite sequence of weights. Then, \( dl((c, w), i) \triangleq \sum_{k=i}^{rs(c, i)-1} w_k \) denotes the delay \( w.r.t. \) \( w \) within which a request \( i \in Rq(c) \) is responded. Also, \( dl((c, w), R) \triangleq \sup_{i \in R} dl((c, w), i) \) is the supremum of all delays of the requests contained in \( R \subseteq Rq(c) \).

### 3 Our contribution

<table>
<thead>
<tr>
<th>( W_n )</th>
<th>Formal definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P</strong></td>
<td>( \forall c \in Cl^\omega, c \in W_n \iff \exists R \subseteq Rq(c),</td>
</tr>
<tr>
<td><strong>PF</strong></td>
<td>( Rq(c) \setminus R \subseteq Rs(c) )</td>
</tr>
<tr>
<td><strong>PP</strong></td>
<td>( \forall (c, w) \in Cl^\omega \times Wg^\omega, (c, w) \in W_n \iff \exists R \subseteq Rq(c),</td>
</tr>
<tr>
<td><strong>FPP</strong></td>
<td>( \exists b \in \mathbb{N}, dl((c, w), Rq(c) \setminus R) \leq b )</td>
</tr>
<tr>
<td><strong>CP</strong></td>
<td>( \forall (c, w) \in Cl^\omega \times Wg^\omega, (c, w) \in W_n \iff \exists R \subseteq Rq(c),</td>
</tr>
<tr>
<td><strong>BCP</strong></td>
<td>( \exists b \in \mathbb{N}, dl((c, w), Rq(c) \setminus R) \leq b )</td>
</tr>
</tbody>
</table>

Table 1: Summary of all winning condition \( W_n \) definitions.

In this contribution we keep working on two-player parity games, under the prompt semantics. We give a clear picture of all different extended parity conditions introduced in the literature working under the prompt assumption and we, also, introduce new parity conditions.

In detail, we introduce and study three new parity conditions named full parity (FP), prompt parity (PP) and full-prompt parity (FPP), respectively. The full parity is defined over colored arenas and requires that all requests must be responded. See Table 1 for a schematic view of this argument. We prove that the complexity of checking whether player \( \exists \) wins under the full parity condition is in PTIME. This result is obtained by a quadratic translation to classic Büchi games. The prompt parity condition, which we consider on both colored and weighted arenas, requires that almost all requests are
responded within a bounded cost, which we name here delay. The full-prompt parity condition is defined accordingly. Observe that the main difference between the cost parity and the prompt parity conditions is that the former is a conjunction of two properties, in each of which a possibly different set of finite requests can be ignored, while in the latter we indicate only one set of finite requests to be used in two different properties. Nevertheless, since the quantifications of the winning conditions range on co-finite sets, we are able to prove that prompt and cost parity conditions are semantically equivalent.

**Theorem 3.1.** Let \( \mathcal{D}_1 = (\hat{A}_1, W_{n_1}, v_0) \) and \( \mathcal{D}_2 = (\hat{A}_2, W_{n_2}, v_0) \) be two games defined on the payoff arenas \( \hat{A}_1 \) and \( \hat{A}_2 \) having the same underlying arena \( A \). Then, player \( \exists \) wins \( \mathcal{D}_2 \) if it wins \( \mathcal{D}_1 \) under the following constraints:

- \( \hat{A}_1 = \hat{A}_2 \) are induced by a weighted arena \( \hat{A} = (A, Cl, Cl, W^\omega, wg) \) and
  1. \((W_{n_1}, W_{n_2}) = (PP, CP)\)
  2. \((W_{n_1}, W_{n_2}) = (CP, PP)\)

**Proof.** The proof of the first item is omitted as trivial. To prove the second item, we show that if a payoff \((c, w) \in Cl^\omega \times W^\omega\) satisfies the CP condition then it also satisfies the PP one. Indeed, by definition, there are a finite set \( R \subseteq Rq(c) \) such that \( Rq(c) \setminus R \subseteq Rs(c) \) and a possibly different finite set \( R' \subseteq Rq(c) \) for which there is a bound \( b \in \mathbb{N} \) such that \( dl((c, w), Rq(c) \setminus R') \leq b \). Now, consider the union \( R'' = R \cup R' \). Obviously, this is a finite set. Moreover, it is immediate to see that \( Rq(c) \setminus R'' \subseteq Rs(c) \) and \( dl((c, w), Rq(c) \setminus R'') \leq b \), for the same bound \( b \). So, the payoff \((c, w)\) satisfies the PP condition, by using \( R'' \) in place of \( R \) in the definition.

Successively, we also prove that the complexity of checking whether player \( \exists \) wins the game under the prompt parity condition is \( \text{UPTIME} \cap \text{COUPTIME} \), in the case of weighted arenas. So, the same result holds for cost parity games and this improves the previously known results. The statement is obtained by a quartic translation to classic parity games. Informally, our algorithm always reduces the original problem to a unique parity game, which is the core of how we gain a better result w.r.t. the time complexity point of view. Obviously, this is different from what is done in [2] as the algorithm there performs several calls to a parity game solver. Observe that, on colored arenas prompt and full-prompt parity conditions correspond to the finitary and bounded-finitary parity conditions [1], respectively. Hence, both the corresponding games can be decided in \( \text{PTIME} \). Our goal is not to make an improvement of the algorithmic complexity but, we prove that checking the winner of a game under all investigated conditions can be done either in \( \text{PTIME} \) or in \( \text{UPTIME} \cap \text{COUPTIME} \). We prove that for full-prompt parity games the \( \text{PTIME} \) complexity holds even in the case the arenas are weighted. Finally, by means of a cubic translation to classic parity games, we prove that bounded-cost parity over weighted arenas is in \( \text{UPTIME} \cap \text{COUPTIME} \), which also improves the previously known result about this condition. We face the computational complexity of solving FP, PP, and BCP games. The technique we adopt is to solve a given game through the construction of a new game over an enriched arena, on which we play with a simpler winning condition. Intuitively, the built game encapsulates in the states of its arena some information regarding the satisfaction of the original condition. To this aim, we introduce the concepts of transition table and its product with an arena. A transition table is an automaton without acceptance
condition. It is used to represent the information of the winning condition mentioned above. Then, the product operation allows to pass this information to the new arena. In general, our constructions are pseudo-polynomial, but if we restrict to the case of having only 0 and 1 as weights over the edges, then they become polynomial, due to the fact that the threshold is bounded by the number of edges in the arena. Moreover, since a game with arbitrary weights can be easily transformed into one with weights 0 and 1, we overall get a polynomial reduction for all the cases. Note that to check if a value is positive or zero is linear in the number of its bits and, therefore, it is linear in the description of its weights.

Finally, Table 2 summarizes the achieved results. In particular, we use the special arrow $\leftarrow$ to indicate that the result is trivial or an easy consequence of another one.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Colored Arena</th>
<th>(Colored) Weighted arena</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity (P)</td>
<td>UPTime $\cap$ CoUPTime $[4]$</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td>Full Parity (FP)</td>
<td>PTime $[3]$</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td>Full Prompt Parity (FPP)</td>
<td>$\leftarrow$</td>
<td>PTime $[3]$</td>
</tr>
</tbody>
</table>

Table 2: Summary of all winning condition complexities.

References


Transformational Termination Analysis of Programs with Pointer Arithmetic

Thomas Ströder (stroeder@informatik.rwth-aachen.de)

LuFG Informatik 2 and AlgoSyn, RWTH Aachen University, Germany

1 Introduction

Traditionally, analysis and synthesis are considered contrary fields of research. In this paper, however, we will show that synergies between these two fields can be exploited to obtain a practically powerful and fully automated method to prove termination of C programs. To this end, we synthesize simple integer transition systems (ITSs) from C programs which capture the essence of the original program’s termination behavior. Then, existing powerful analysis techniques for this simple mathematical formalism can be used to prove termination of real C programs automatically. As a byproduct, we also prove memory safety of the original program during the transformation. In particular, the presented approach can deal with explicit pointer arithmetic as commonly seen, e.g., in string algorithms.

For instance, the following standard C implementation of strlen [9, 15] computes the length of the string at pointer str. In C, strings are usually represented as a pointer to the heap, where all following memory cells up to the first one that contains the value 0 are allocated memory and form the value of the string.

\[
\text{int strlen(char* str) \{char* s = str; while(*s) s++; return s-str;\}}
\]

The difficulty in analyzing such algorithms for termination is that the control flow depends on the content of the memory and that arbitrary memory addresses can be accessed using pointer arithmetic. Moreover, access to non-allocated memory is undefined behavior in C. Thus, for a sound termination analysis, we also have to prove memory safety (i.e., absence of accesses to non-allocated memory) of the program in question. Fortunately, this can be proven “along the way” when transforming the original program to an integer transition system.

Intuitively, the reason why the strlen algorithm above is memory safe and terminating is that there is some address end \(\geq\) str (an integer property of end and str) such that *end is 0 (a pointer property of end) and all addresses str \(\leq s \leq\) end are allocated.

To improve the generality of our method and to avoid some platform-dependent intricacies of C, we analyze programs in the platform-independent intermediate representation (IR) of the LLVM compilation framework [8].

*Supported by DFG grant GI 274/6-1 and Research Training Group 1298 (AlgoSyn).
Our approach consists of three stages: First, an abstract interpretation [4] of the program over-approximating its behavior is computed. If this over-approximation does not contain violations of memory safety, this already proves memory safety of the original program. Second, from the integer and pointer properties found in the abstract interpretation, we synthesize ITSs with the property that termination of the synthesized systems implies termination of the over-approximation (and, thus, also of the original program). Third, existing techniques for termination analysis of ITSs [10] are employed to prove termination of the synthesized transition systems. This approach has been implemented in the termination prover AProVE [5], which (using the presented approach) won the termination competition (termCOMP) 2014 [13] on C programs and the termination category of the software verification competition (SV-COMP) 2015 [12].

The content of this paper is joint work with J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, and P. Schneider-Kamp. It is a short version of [11], in which further details can be found.

2 Approach

The strlen program from the previous section is compiled by the Clang compiler [3] to the LLVM program on the right. To ease readability, we wrote variables without "%" in front (i.e., we wrote "str" instead of "%str" as in proper LLVM) and added line numbers.

8-bit characters have the type i8 in LLVM. Pointer types have the suffix *, so str has the type i8*. LLVM programs are organized by basic blocks. In this example, we have the basic blocks entry, loop, and done. The instructions load, icmp eq (integer compare equal), sub (subtract), and ret (return) have the default intuitive meaning. The instruction ptrtoint is a cast from a pointer type to an integer. For branching depending on the truth value of a variable, the instruction br is used. Its arguments are the boolean variable and the two labels reached if the variable is true or false, respectively. Pointer arithmetic is performed using the getelementptr instruction. Its first argument is a base pointer, its second argument is an offset. It yields the pointer to the base address plus the offset. The phi instruction assigns a value to a variable at the beginning of a basic block depending on the block that the control flow was in before the current block.

Intuitively, entry corresponds to the C code before the while-loop and to the first check of the loop condition. The loop block corresponds to the loop body followed by the check of the loop condition. Finally, the done block corresponds to the C code after the while-loop.

For this LLVM program, we obtain the (simplified) abstract interpretation in Fig. 1. The components of states in this graph are first the program position, second the values of

```
define i32 @strlen(i8* str) {
entry: 0: c0 = load i8* str
    1: c0zero = icmp eq i8 c0, 0
    2: br i1 c0zero, label done, label loop
loop: 0: olds = phi i8* [str,entry],[s,loop]
    1: s = getelementptr i8* olds, i32 1
    2: c = load i8* s
    3: czero = icmp eq i8 c, 0
    4: br i1 czero, label done, label loop
done: 0: sfin = phi i8* [str,entry],[s,loop]
    1: sfinint = ptrtoint i8* sfin to i32
    2: strint = ptrtoint i8* str to i32
    3: size = sub i32 sfinint, strint
    4: ret i32 size }
```
Transformational Termination Analysis of Programs with Pointer Arithmetic

![Diagram]

Figure 1: Abstract interpretation for strlen

the program variables, third a set of relations known to hold in the state, and fourth known contents of the memory. A program position consists of the previous basic block (needed to evaluate phi instructions; ε denotes no basic block), the current basic block, and the line number within the current basic block. We assign abstract values (so-called references) to the program variables. Thus, we can express knowledge about these references in the set of relations. The knowledge that at the address a we have the content c is denoted $a \rightarrow c$. The special predicate alloc(x, y) denotes that the memory between addresses x and y (both inclusive) is allocated. In particular, this implies $x \leq y$. The obtained graph is always finite and over-approximates all possible runs of the original program (which are infinitely many in most cases). To obtain finite graphs, we have to find existing states that subsume states reached later. To make sure that this is always possible, we also have to generalize states. The rules used to automatically obtain such an abstract interpretation from an LLVM program are given in [11].

The first state (labeled A) has the intuitive meaning that we are at the very first instruction and know that the parameter str points to a string, i.e., to an address $u_{str}$, from which an allocated memory area starts and ends at the address $v_{end}$ containing the value 0.

Since this graph does not contain any state where non-allocated memory is accessed, memory safety of strlen is proven. Now to prove termination, we only need to consider non-trivial strongly connected components (SCCs) in the graph as each infinite run must eventually stay in an SCC. For each non-trivial SCC, an integer transition system is synthesized. In this graph, there is only one non-trivial SCC from the node labeled $B$ over $C$ to $D$ and from there back to $B$.

The synthesis of integer transition systems works as follows. Each edge within an SCC is translated to a transition rule. The states of the integer transition system are the states
of the SCC, but states with an edge back to an already existing state (so-called instance edges) are identified with the state to which they refer back. In our example, we have the states $B$ and $C$ ($D$ is identified with $B$). The integer variables in the transition system are the references in the states of our abstract interpretation and the conditions on the transition rules are the relations known in the states associated with the corresponding edge (but only those which do not contain references not assigned to any program variable and not being a known memory address). In case of an outgoing instance edge at the target state, we also add equations between references assigned to the same program variable in the two states associated with the instance edge. The transition rules have the form $\langle \text{state}\rangle(\text{variables}) \xrightarrow{\text{condition}} \langle \text{state}\rangle(\text{variables})$. So here we obtain the two rules:

\[
\begin{align*}
B&(v_{\text{str}}, v_c, v_s, v_{\text{olds}}, v_{\text{end}}, \ldots) \quad \rightarrow \quad C(v_{\text{str}}, w_c, w_s, w_{\text{olds}}, v_s, v_{\text{end}}, \ldots) \\
&\quad \text{with:} \\
&v_c \neq 0, \\
&v_s = v_{\text{olds}} + 1, \\
&v_{\text{olds}} \geq v_{\text{str}}, \\
&v_s < v_{\text{end}}, \\
&w_s = w_{\text{olds}} + 1, \\
&w_{\text{olds}} = v_s, \\
&C&(v_{\text{str}}, w_c, w_s, w_{\text{olds}}, v_s, v_{\text{end}}, \ldots) \quad \rightarrow \quad B(v_{\text{str}}, v_c, v_s, v_{\text{olds}}, v_{\text{end}}, \ldots) \\
&\quad \text{with:} \\
&w_s = w_{\text{olds}} + 1, \\
&w_c \neq 0, \\
&w_c = v_c, \\
&w_s = v_s, \\
&w_{\text{olds}} = v_{\text{olds}}, \\
\end{align*}
\]

This can be simplified automatically to the following one-rule system:

\[ f(x, y) \xrightarrow{x \leq y} f(x + 1, y) \]

This system captures exactly the reason why strlen terminates: There is a “current” address $x$, which is increased until it reaches another address $y$ which stays the same (and at which the value 0 is stored, but this knowledge is irrelevant for the termination proof). Virtually all termination provers of integer transition systems can prove termination of the resulting system within less than a second. Thus, termination of strlen has been proven.

### 3 Related Work, Experiments, and Conclusion

There exists a plethora of tools and techniques for termination analysis of imperative programs. We refer to our competitors [2, 6, 7, 14] in the two competitions [12, 13] as they reflect a wide range of the most recent developments in this field.

In addition to these two competitions, we also conducted experiments ourselves to evaluate the practical power of AProVE. The results of this evaluation can be found in [1].

Apart from AProVE’s success at the competitions, it is noteworthy that AProVE is the first and currently one of the two only tools capable of analyzing C programs involving pointer arithmetic for termination (including memory safety) fully automatically (the other tool is Ultimate Automizer [6]).
Transformational Termination Analysis of Programs with Pointer Arithmetic

References


1 Introduction

While Moore’s law is still upheld by increasing the number of cores of processors, the construction of concurrent programs that exploit the added computational capacity has become significantly more complicated. This holds particularly true for debugging multi-threaded shared-memory software: unexpected interactions between threads may result in erroneous and seemingly non-deterministic program behavior whose root cause is difficult to analyze. Therefore, concurrency bugs which are on the rise are among the most difficult software bugs to detect and diagnose. Previous methods mostly have focused on targeting a single type of concurrency bugs, such as data races (concurrent conflicting accesses to the same memory location) [7] and atomicity violations (an interference between supposedly indivisible critical regions) [2]. We propose a mining-based technique to explain concurrency bugs that is oblivious to the nature of the specific bug, thus providing a general framework for concurrency bug explanation. Moreover, unlike some general techniques such as [6], our mining-based method does not rely on any given pattern templates. We assume that we are given a set of concurrent execution traces, each of which is classified as passed or failed. This is a reasonable assumption, as this is a prerequisite for systematic software testing. Although the traces of concurrent programs are lengthy sequences of events, only a small subset of these events is responsible for the bug, thus sufficient to explain an erroneous behavior. In general, these events do not occur consecutively in the execution trace, but rather at an arbitrary distance from each other. Therefore, we use a data mining technique called sequential pattern mining to isolate ordered sequences of non-contiguous events which occur frequently in failing traces and then rank them based on the number of their occurrences in passing traces. We consider the highly ranked sequences of events that occur frequently only in failing traces an explanation of the system failure, as they can reveal its causes in the execution traces. Since the scalability of sequential pattern mining is limited by the length of the traces, we present a novel abstraction technique which reduces the length of the traces as well as the number of events by mapping sequences of concrete events to single abstract events. We show that this abstraction step preserves all original behaviors while reducing the number of patterns to consider. However, as an artifact of the abstraction, spurious patterns are produced. Spurious as well as misleading explanations are then eliminated by a subsequent filtering step, helping the programmer to focus on likely causes of the failure.
2 Preliminaries

In a multi-threaded program comprising a set of shared variables and a number of threads, interaction between the threads is done via the read and write accesses to shared variables. In our setting, we assume threads are sequentially correct, thus concurrency bugs are due to problematic interactions between the threads and manifest themselves in problematic accesses to shared variables. We therefore explain concurrency bugs by isolating problematic accesses to shared variables from the concurrent execution traces.

A concurrent execution trace is defined as a totally ordered finite sequence of events corresponding to some interleaving of instructions from the threads. We call executions leading to a failure (a behavior contradicting the specification) failing or bad, and all other executions passing or good executions. For our bug explanation purposes, we log only the order of read and write events for shared variables. Two such events conflict if they are issued by different threads, access the same variable, and at least one of them is a write. Given two conflicting events, we distinguish three cases of data dependency: (a) flow-dependence: one event reads a value written by the other, (b) anti-dependence: one event reads a value before it is overwritten by the other, and (c) output-dependence: both events write the same variable.

2.1 Bug Explanation Patterns

We illustrate via an example how a deviation of the accesses to and the data-dependencies of a shared variable causes a failure. Figure 1 shows a code fragment that non-atomically updates the balance of a bank account (stored in the shared variable balance) at locations $\ell_1$ and $\ell_2$. The example does not contain a data race, since balance is protected by the lock balance_lock. In the failing execution at the very left of Figure 1 balance is overwritten with a stale value in thread 1, “killing” the transaction of thread 2 that writes balance, therefore the final value of this variable becomes inconsistent with its expected value. This is reflected by the data-dependencies between the highlighted write/read events in the failing trace. This combination of events and the corresponding dependencies do not arise in any passing trace, since no context switch occurs between the events $R_1(balance)$ and $W_1(balance)$. Accordingly, the sequence of events highlighted in the left trace in Figure 1 in combination with the dependencies reveals the problematic memory accesses to balance. We refer to this sequence as a bug explanation pattern. We emphasize that the events belonging to this pattern do not occur consecutively inside the trace, but are interspersed with other unrelated events. In general, events belonging to a bug explanation pattern can occur at an arbitrary distance from each other due to scheduling.
explanations are therefore, in general, subsequences of execution traces.

3 Mining Bug Explanation Patterns

We use a data mining technique called sequential pattern mining to identify bug explanation patterns. Given a minimum support threshold, sequential pattern mining is a technique to extract frequent subsequences or sequential patterns from a dataset [5]. In our setting, we are interested in subsequences occurring frequently in the sets of passing (good) and failing (bad) execution traces, respectively. Intuitively, bug explanation patterns occur more frequently in the bad dataset. To extract bug explanation patterns, we first mine frequent subsequences with a given minimum support threshold from the bad dataset. We then determine which of them are frequent only in the bad dataset but not in the good dataset by computing the number of their occurrences in the good dataset. Accordingly, we rank the patterns such that patterns which occur more frequently in the bad dataset are ranked higher.

4 Abstracting Execution Traces

With increasing length of the execution traces and number of events, sequential pattern mining quickly becomes intractable [1]. To alleviate this problem, we introduce macro-events that represent events of the same thread occurring consecutively inside an execution trace. By defining macros, we obtain a more compact representation of a set of execution traces by mapping every trace to its corresponding macro trace. For example, for a defined set of macros $M = \{ m_0 \overset{\text{def}}{=} \langle e_0, e_2 \rangle, m_1 \overset{\text{def}}{=} \langle e_1, e_2 \rangle, m_2 \overset{\text{def}}{=} \langle e_3 \rangle, m_3 \overset{\text{def}}{=} \langle e_4, e_5, e_6 \rangle, m_4 \overset{\text{def}}{=} \langle e_8, e_9 \rangle, m_5 \overset{\text{def}}{=} \langle e_5, e_6, e_7 \rangle \}$ and the traces $\sigma_1$ and $\sigma_2$ as defined below, we obtain

$$
\sigma_1 = \langle e_0, e_2, e_4, e_5, e_6, e_8, e_9 \rangle \quad \text{macro}(\sigma_1) = \langle m_0, m_2, m_3, m_4 \rangle \\
\sigma_2 = \langle e_1, e_2, e_5, e_6, e_7, e_3, e_8, e_9 \rangle \quad \text{macro}(\sigma_2) = \langle m_1, m_5, m_2, m_4 \rangle
$$

This transformation reduces the number of events as well as the length of the traces while preserving the context switches, but hides information about the frequency of the original events. For the example (1), $m_3$ and $m_5$ occur once in macro traces, even though the events $\{ e_5, e_6 \}$ shared by them occur twice in the original traces. While this problem can be amended by refining $M$ by adding $m_6 = \langle e_5, e_6 \rangle$, $m_7 = \langle e_4 \rangle$, and $m_8 = \langle e_6 \rangle$, for instance, this increases the length of the trace and the number of events, counter our original intention. Instead, we introduce an abstraction function $\alpha : M \rightarrow A$ which maps macros to a set of abstract events $A$ according to the events they share. The abstraction guarantees that if $m_1$ and $m_2$ share events, then $\alpha(m_1) = \alpha(m_2)$. For the example above (1), we obtain, for instance, $\alpha(m_0) = \alpha(m_1) = \{ m_0, m_1 \}$ and $\alpha(m_3) = \alpha(m_5) = \{ m_3, m_5 \}$. The corresponding abstract trace of $\sigma_1$ is then $\alpha(\text{macro}(\sigma_1)) = \{ \langle m_0, m_1 \rangle, \langle m_2 \rangle, \langle m_3, m_5 \rangle, \langle m_4 \rangle \}$. Since when two macros share an event they are mapped to the same abstract event, it is guaranteed that frequency of concrete events in the original traces is preserved. Therefore, it can be shown that the patterns mined from abstract traces over-approximate the patterns of the corresponding original execution traces. Note that even though the abstract pattern is significantly shorter, the number of context switches is the same. While our abstraction preserves the original patterns, it may introduce spurious patterns which do not occur in any original traces. We filter spurious patterns by mapping them to the original traces.
Table 1: Length reduction results by abstracting the traces

| Prog. Category | Name                  | $|\Sigma_B|$ | $|\Sigma_G|$ | Min. Trace Len. | Max. Abst. Trace Len | Len Red. |
|----------------|-----------------------|-------------|-------------|-----------------|----------------------|---------|
| Synthetic      | BankAccount           | 40          | 5           | 178             | 13                   | 93%     |
|                | CircularListRace      | 64          | 6           | 184             | 9                    | 95%     |
|                | WrongAccessOrder      | 100         | 100         | 48              | 20                   | 58%     |
| Bug Kernel     | Apache-25520(Log)     | 100         | 100         | 114             | 16                   | 86%     |
|                | Moz-jsStr             | 70          | 66          | 404             | 18                   | 95%     |
|                | Moz-jsInterp          | 610         | 251         | 430             | 101                  | 76%     |
|                | Moz-txtFrame          | 99          | 91          | 410             | 57                   | 86%     |

Table 2: Mining results

<table>
<thead>
<tr>
<th>Program</th>
<th>min_supp</th>
<th>$#\alpha$</th>
<th>$#\gamma$</th>
<th>$#feas$</th>
<th>$#filt$</th>
<th>$#rs = 1$</th>
<th>$#grp$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankAccount</td>
<td>100%</td>
<td>65</td>
<td>13054</td>
<td>19</td>
<td>10</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>CircularListRace</td>
<td>95%</td>
<td>12</td>
<td>336</td>
<td>234</td>
<td>18</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>WrongAccessOrder</td>
<td>100%</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache-25520(Log)</td>
<td>100%</td>
<td>160</td>
<td>1650</td>
<td>667</td>
<td>16</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Moz-jsStr</td>
<td>100%</td>
<td>83</td>
<td>615056</td>
<td>486</td>
<td>90</td>
<td>76</td>
<td>4</td>
</tr>
<tr>
<td>Moz-jsInterp</td>
<td>100%</td>
<td>83</td>
<td>279882</td>
<td>49</td>
<td>23</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>Moz-txtFrame</td>
<td>90%</td>
<td>1192</td>
<td>5137</td>
<td>2314</td>
<td>200</td>
<td>32</td>
<td>11</td>
</tr>
</tbody>
</table>

Sequential pattern mining ignores the underlying semantics of the events and macros. This has the undesirable consequences that we obtain numerous patterns that are not explanations. Accordingly, we define a set of constraints to eliminate misleading patterns. Patterns must contain events of at least two different threads (since we are interested in concurrency bugs). We restrict our search to patterns with a limited number (at most 4) of context switches, since there is empirical evidence that real world concurrency bugs involve only a small number of threads, context switches, and variables [3]. We require that for each macro in a pattern there is a data-dependency with at least one other macro in the pattern (by lifting the data-dependencies introduced in Section 2 to macros).

5 Experimental Evaluation

To evaluate our approach, we present 7 case studies listed in Table 1 (6 of them are taken from [4]). The programs are bug kernels capturing the essence of bugs reported in Mozilla and Apache, or synthetic examples created to cover a specific bug category. We generate execution traces using the concurrency testing tool INSPECT [10], which systematically explores all possible interleavings for a fixed program input. In Table 1, the last column shows the length reduction (up to 95%) achieved by means of abstraction. State-of-the-art sequential pattern mining algorithms are typically applicable to sequences of length less than 100 [9, 5]. Therefore, the reduction of the original traces is crucial. The results of mining for the given programs are provided in Table 2. The column labeled min_supp shows the support threshold required to obtain at least one bug explanation pattern (lower thresholds yield more patterns). For the given value of min_supp, the table shows the number of resulting patterns after abstraction (#\(\alpha\)), concretization (#\(\gamma\)), filtering spurious patterns (#feas), and filtering misleading patterns (#filt). The number
of patterns which only occur in the bad dataset is given in column 7. Finally, we group
the resulting patterns according to the set of data-dependencies they contain; column
#grp shows the resulting number of groups. We verified manually that all groups which
occur only in the bad dataset are an adequate explanation of at least one concurrency
bug in the corresponding program. One pattern from these groups for Moz-jsStr case

<table>
<thead>
<tr>
<th></th>
<th>101</th>
<th>102</th>
<th>103</th>
<th>104</th>
<th>105</th>
<th>106</th>
<th>107</th>
<th>108</th>
<th>109</th>
<th>110</th>
<th>111</th>
<th>112</th>
<th>113</th>
<th>114</th>
<th>115</th>
<th>116</th>
<th>117</th>
<th>120</th>
<th>121</th>
<th>122</th>
<th>123</th>
<th>124</th>
<th>276</th>
<th>277</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1-R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2-W1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>totalStrings</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lengthSum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: A bug explanation pattern: three macros, numbers represent the ids of events
study is given in Figure 2. The pattern and the data-dependencies reveal this atomic-
ity violation: the values of totalStrings and lengthSum read by thread 2 are inconsistent
due to a context switch that occurs between the updates of these two variables by thread 1.

References

In *Model Checking and Software Verification (SPIN)*, 2012.

access interleaving invariants. In *Architectural Support for Programming Languages


localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Val-

[7] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas And-

traces to explain concurrency bugs. In *Runtime Verification (RV)*, LNCS, pages

[9] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large
datasets. In *Proceedings of 2003 SIAM International Conference on Data Mining
(SDM’03)*, 2003.

dynamic partial order reduction based verification of threaded software. In *Model
Checking and Software Verification (SPIN)*, pages 58–75. LNCS, 2007.
1 Introduction

A common problem in current applications are missing or unreliable information. At the point in time when optimization decisions have to be made not all data is present, possibly because it does not exist yet. This occurs in airplane overbooking, where it is unclear at every point in time before take-off how many people that booked a ticket will actually show up. In the assignment of students onto seminar slots or dorms, where you do not know how many students will ask for a place and how long they will actually stay. Or in online ad allocation. Here it is unknown to the allocator which search requests will be typed in in the future, so he can only guess what ad he should show in response to the current query.

We analyze this type of problems through the lens of online algorithms. To be specific, we work under the random order model. Here it is assumed that the problem instance is generated adversarially, but the order in which the input arrives is randomly permuted. A classical example for this model is the secretary problem. Assume you want to hire a secretary and you have invited a dozen candidates to an interview and they are waiting outside the door. Now furthermore assume that after each interview you have to decide if you want to take the candidate or not. If you decline he will walk away and not be available anymore. Now you obviously want the best candidate for the position, but you can only hire one. The algorithm that maximizes the probability of picking the best candidate is really simple and will be presented in section 2 as a special case. Please note that this algorithm has a probability of $\frac{1}{e}$ to pick the best candidate and that it is impossible to get a higher chance.

A direct generalization of the secretary problem is the online weighted bipartite matching problem. Here the input is an edge-weighted bipartite graph and we optimize towards a maximum weight matching. Remarkably we can show that the optimal algorithm for this problem achieves the same competitive guarantee as the known optimal algorithm for the secretary problem. In fact the algorithm is a direct generalization of the known algorithm, but a new approach is required for the analysis.

Another even more general problem that we consider in the random order model are linear packing programs. Here we have a $(1 - \epsilon)$-competitive algorithm if the space in all constraints is large enough compared to the size of the items.

*Supported by the DFG GRK/1298 “AlgoSyn”.*
Additionally to these two fundamental results we can adapt our algorithms for all kinds of special cases. We extend the matchings towards hypermatchings and combinatorial auctions. We consider variations of linear packing problems like the knapsack problem or the generalized assignment problem. For all these problems we describe and analyze online algorithms in the random order model. These algorithms all follow similar design principles, a design framework that is described in section 2. Please note that, while the algorithms all fit a certain framework, the analysis unfortunately is not straight forward from one problem to the other.

2 Algorithm Framework

For all these problems we derive algorithms that work similarly to the classical secretary algorithm. Usually there is some form of sampling to collect information in the beginning. Then for every item that arrives online we compute an optimal solution on all items that are known up to this point in time. We call this intermediate solution local solution. Then we assign the current online item in them same way it is allocated in the local solution, if this allocation is feasible with respect to the current global solution.

For online weighted bipartite matching the algorithm is practically identical to the well-known optimal algorithm for the secretary problem [1]. One side of the bipartite graph is known before the algorithm starts, we call this set of vertices $R$ and the side that arrives online $L$.

Algorithm 1: Bipartite online matching

| Input | vertex set $R$ and cardinality $n = |L|$ |
|-------|--------------------------------------|
| Output: matching $M$ |

Let $L'$ be the first $\lfloor n/e \rfloor$ vertices of $L$; // sampling phase

$M := \emptyset$;

for each subsequent vertex $\ell \in L - L'$ do // steps $\lceil n/e \rceil$ to $n$

$L' := L' \cup \ell$;

$M^{(\ell)} :=$ optimal matching on $G[L' \cup R]$; // e.g. by Hungarian method

Let $e^{(\ell)} := (\ell, r)$ be the edge assigned to $\ell$ in $M^{(\ell)}$;

if $M \cup e^{(\ell)}$ is a matching then

add $e^{(\ell)}$ to $M$;

If the known side $R$ of the graph has cardinality 1, then the problem corresponds to the secretary problem. We have shown that this algorithm is $e$-competitive for weighted bipartite matching. This is optimal since it has been shown that there is no better algorithm for the secretary problem either.

In the analysis we simply sum up the expected value of the online allocation made in each round. This is straight forward after one key observation. The expected value of the online allocation in a step is independent of the order in which those the items up to this point arrived. This allows us to unravel the stochastic process from the last step of the algorithm. Since the expected value in any step is independent of the order up to this point we can simply draw one of the items that are available in this step uniformly
at random to be the current online item. In the analysis we do this iteratively from the last step to the beginning. We refer to the original paper [1] for the detailed proof.

We apply similar techniques in our algorithm for online packing linear programs [2]. In this problem there is an adversarially created linear program that only contains packing constraints. Formely we optimize the target function \( \max c^T x \) such that \( Ax \leq b \) and \( 0 \leq x \leq 1 \) for \( c_j, a_{i,j}, b_i \in \mathbb{R}_+ \). In the online version that we consider, the columns are revealed online over time in small batches. Whenever such a group of columns arrives the algorithm can pick one of the associated variables and set it to a non-zero value. Again we use the random order model, thus we assume that the columns do not arrive in adversarial order, but in an order randomly permuted by an uniformly drawn random permutation.

**Algorithm 2: Online packing LP**

Let \( S \) be the index set of known requests, initially \( S := \emptyset \);
Set \( y := 0 \);

for each arriving request \( j \) do  
   // steps \( \ell = 1 \) to \( n \)
   Set \( S := S \cup \{j\} \) and \( \ell := |S| \);
   Let \( \tilde{x}^{(\ell)} \) be an optimal solution of the LP \( \max c^T x \) s.t. \( Ax \leq \frac{\ell}{n}b \), \( 0 \leq x \leq 1 \), \( x_j = 0 \) (\( \forall j' \notin S \));
   Choose an option \( k^{(\ell)} \) (possibly none) where option \( k \) has probability \( \tilde{x}_{j,k}^{(\ell)} \);
   // rand. rounding
   Define \( x^{(\ell)} \) with \( x_{j',k}^{(\ell)} = \begin{cases} 1, & \text{if } j' = j \text{ and } k = k^{(\ell)}; \\ 0, & \text{otherwise}; \end{cases} \) // tentative allocation
   if \( A(y + x^{(\ell)}) \leq b \) then
      // feasibility check
      Set \( y := y + x^{(\ell)} \);  
      // online allocation

Here the sampling phase is not as explicit as in the bipartite matching algorithm. We start allocating value to the variables right away, but we scale down the capacities in the packing constraints. Therefore the probability to allocate weight to the current online variables starts out low and increases over the online process. This is crucial for our main results on linear packing programs. We show that algorithm 2 is \((1 - \epsilon)\)-competitive if the capacities on the constraints are large enough. Specifically we require the capacities to be in \( \Omega\left(\frac{\log(d)}{\epsilon^2}\right) \) if the weights on the variables are in \([0, 1]\). Here \( d \) is a sparsity measure for the constraint matrix the maximum number of non-zero entries in any column. This result is tight. It has been shown that there is no \((1 - \epsilon)\)-competitive algorithm for smaller capacities.

Furthermore we show that the algorithm does not degrade when the capacities are smaller. Specifically the algorithm still is \( O\left(d^{1/(b-1)}\right)\)-competitive for any capacity \( b \geq 2 \).

**References**

Derivatives of WS1S Formulas

Dmitriy Traytel (traytel@in.tum.de)
Fakultät für Informatik, Technische Universität München, Germany

In his seminal work [5], Büchi envisioned weak monadic second-order logic of one successor (WS1S) to become a “more conventional formalism [that] can be used in place of regular expressions […] for formalizing conditions on the behaviour of automata”. This vision became truth—WS1S has been used to encode decision problems in hardware verification [1], network verification [2], synthesis [6], as well as many others.

Equivalence of WS1S formulas is decidable, although the decision procedure’s complexity is non-elementary [9]. Nevertheless, the MONA tool [7] shows that the daunting theoretical complexity can often be overcome in practice by employing a multitude of smart optimizations. Similarly to Büchi, MONA’s user manual [8] calls WS1S a “simple and natural notation” for regular languages.

Traditionally, decision procedures for WS1S do not try to benefit themselves from the conventional, simple, and natural logical notation. Instead, by exploiting the logic-automaton connection, formulas are translated into finite automata which are then minimized. During the translation all the rich algebraic formula structure including binders and high-level constructs is lost. On the other hand, the subsequent minimization might have benefited from some simplifications on the formula level.

Concerning the algebraic structure, regular expressions are situated somewhere in between of WS1S formulas and finite automata. In earlier work [15], we propose a semantics-preserving translation of WS1S formulas into regular expressions. Thereby, equivalence of WS1S formulas is reduced to equivalence of regular expressions. To decide equivalence of regular expressions, we employ a coalgebraic decision procedure based on a finality test and Brzozowski derivatives [4]—the coalgebra structure on regular expressions [12].

In recent work [13], we go one step further by defining a coalgebra structure directly on WS1S formulas. The main contributions are:

• We define a symbolic derivative operation for a WS1S formula.
• We define a finality test determining if a formula holds in the empty interpretation.
• Taking the two above notions together, we obtain a decision procedure for WS1S that operates only on formulas.
• We formalize the newly defined notions in Isabelle/HOL [10] and formally verify that the obtained algorithm indeed decides equivalence of WS1S formulas.

The obtained decision procedure can be considered an elegant toy—implementable only with a few hundreds lines of Standard ML [14] and teachable in class. By no means it should be evaluated against MONA’s thousands of lines of tricky performance optimizations. On the other hand, we are confident that symbolic decision procedures must not hide behind automata-based ones in terms of performance in general as witnessed by several successful examples [3, 11].
References


Probabilistic Logic and Regular Expressions on Finite Trees

Thomas Weidner∗(weidner@informatik.uni-leipzig.de)

Institut für Informatik, Universität Leipzig, Germany

Introduction

Probabilistic tree automata (PTA) are a well-known formalism with a wide range of applications. A prominent use is natural language processing. In this work we connect PTA to two other classical concepts: Monadic second order logic and regular expressions. Both concepts enjoy vast success in nearly all fields of theoretical computer science. Our goal is to transfer the theorem of Büchi and Elgot and the theorem of Kleene to the setting of probabilistic tree automata. We consider the quantitative behaviour of automata, formulas and expressions: A function mapping trees to probability values.

For the cases of finite and infinite words several similar results already exist: The expressive equivalence between probabilistic (Müller) automata and probabilistic MSO was shown in [4]. In [1] probabilistic regular expressions for finite words have been defined, which have the same expressive power as probabilistic automata\(^1\). This result has been extended to infinite words in [5].

Probabilistic Regular Expressions

Let \( Σ = (Σ_n)_{n≥0} \) be a finite rank alphabet and \( T_Σ \) the set of all trees over \( Σ \). We extend the rank alphabet by an additional finite set of variables \( V \) and write \( T_Σ(V) \) for the set of all trees where symbols from \( V \) may appear as additional leaf labels. First, we define the operations used in the expressions for general probabilistic tree functions: Let \( f,g : T_Σ(V) \to [0,1] \), \( p \in [0,1] \) and \( z ∈ V \). Define

\[
\begin{align*}
(p \cdot f)(t) &= p \cdot f(t) \\
(f \cdot z g)(t) &= \sum_{s ∈ T_Σ(V)} f(s) \cdot \prod_{p ∈ \text{pos}_z(s)} g(t|_p) \\
(f + g)(t) &= f(t) + g(t) \\
f^∞ z(t) &= \lim_{n \to ∞} f^{z^n}(t)
\end{align*}
\]

where \( t[M \leftarrow z] \) is the tree obtained by replacing the subtrees at every position in \( M \) by the symbol \( z \), and \( t|_p \) is the subtree of \( t \) below \( p \). The product operation \( f \cdot z g \) is the same

\*Partly supported by DFG Graduiertenkolleg 1763 (QuantLA)
\(^1\)They only considered probabilistic automata where every final state is a sink state.

261
The “infinity-iteration” can be seen as a deterministic variant of the Kleene-star: In the computation of the Kleene-star $f^\star z$ there is a choice every time a leaf labelled $z$ occurs, whether to continue the iteration and do a substitution at this position, or to stop the iteration. When computing $f^\infty z$ there is no such choice, the computation has to continue doing substitutions until no leaf is labelled by $z$.

Note that the resulting function of these operations may not be well-defined, i.e. attains values outside $[0, 1]$ or the limit does not exist at all. Therefore we do not allow arbitrary combinations of these operations but use a restricted syntax.

The syntax of probabilistic regular tree expressions is given by

$$E ::= z | \sum_{f \in \Sigma} f(E, \ldots, E) \mid pE + (1 - p)E \mid E \cdot z \mid E^\infty z.$$  

This restricted syntax does not allow for distributivity laws to hold. Thus we additionally close the set of expressions under the known distributivity laws. The behaviour of an expression $E$ is a function $\|E\| : T_\Sigma(V) \to [0, 1]$. For expressions of the form $f(E_1, \ldots, E_n)$ the behaviour is given by

$$\|f(E_1, \ldots, E_n)\|(t) = \begin{cases} 
\prod_{i=1}^{n} \|E_i\|(t_i) & \text{if } t = f(t_1, \ldots, t_n) \\
0 & \text{otherwise.} 
\end{cases}$$

For expressions of all other forms, the behaviour directly translates to the operations given above.

Before we state our equivalence theorem, we recall the definition of PTA: A probabilistic (top-down) tree automaton $A$ is a tuple $(Q, \delta, \mu, F)$ where $Q$ is a finite, non-empty set of states, $\delta = \bigcup_{n \geq 1} \delta_n$, where $\delta_n : Q \times \Sigma_n \to \Delta(Q^n)$, is the transition probability function, $\mu \in \Delta(Q)$ the initial probabilities, and $F \subseteq Q \times \Sigma_0$ the accepting condition. The behaviour $\|A\|$ of $A$ is given by

$$\|A\|(t) = \sum_{(\rho, (\rho(x), t(x))) \in F} \mu(\rho(\varepsilon)) \prod_{x \in \text{inner}(t)} \delta(\rho(x), t(x))(\rho(x 1), \ldots, \rho(x n_x)),$$

where $n_x$ is the number of children of $x$, $\text{pos}(t)$ is the set of all positions in $t$, $\text{leaf}(t)$ the set of all leaf positions, and $\text{inner}(t)$ the set of all inner positions.

**Theorem 1** Let $f : T_\Sigma \to [0, 1]$. The following statements are equivalent:

1. $f = \|A\|$ for some probabilistic top-down tree automaton $A$
2. $f = \|E\|$ for some probabilistic regular tree expression $E$

Both translations are effective.

**Probabilistic MSO Logic**

To obtain a probabilistic extension of classical MSO logic we add an additional second order “expected value” quantifier to the logic and close under complement and conjunction. Formally, let the syntax of a PMSO formula $\varphi$ in BNF be given by

$$\varphi ::= \psi \mid \varphi \land \varphi \mid \neg \varphi \mid E_p X.\varphi,$$
where $\psi$ is a Boolean MSO formula, $p \in [0,1]$ is a probability value, and $X$ is a second order variable. The semantics of PMSO formulas are defined inductively. Given a tree $t$ and an assignment of variables $\alpha$, we define

$$\llbracket \psi \rrbracket (t, \alpha) = \begin{cases} 1 & \text{if } (t, \alpha) \models \psi \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket \varphi_1 \land \varphi_2 \rrbracket (t, \alpha) = \llbracket \varphi_1 \rrbracket (t, \alpha) \cdot \llbracket \varphi_2 \rrbracket (t, \alpha)$$

$$\llbracket \neg \varphi \rrbracket (t, \alpha) = 1 - \llbracket \varphi \rrbracket (t, \alpha)$$

$$\llbracket E_p X. \varphi \rrbracket (t, \alpha) = \sum_{M \subseteq \text{pos}(t)} \llbracket \varphi \rrbracket (w, \alpha[X \mapsto M]) \cdot p[M](1 - p)[\text{pos}(t) \setminus M].$$

The intuition of $E_p X. \varphi$ is as follows: For every position of the tree one tosses an unfair coin and depending on the outcome this position is included in the set or not. That is, all positions are independent and identically distributed. Using the so constructed probability distribution on the subsets of positions, one computes the expected semantics of $\varphi$.

Unfortunately, as in the classical setting, probabilistic top-down tree automata do not capture the whole expressive power of probabilistic MSO. Therefore, we resort to bottom-up tree automata. We say $A = (Q, \delta, F)$ is a probabilistic bottom-up tree automata if $Q$ is a finite, non-empty set of states, $F \subseteq Q$ a set of final states and $\delta = \bigcup_{n \geq 0} \delta_n$, with $\delta_n : Q^n \times \Sigma_n \to \Delta(Q)$, is the transition probability function. Note that $\delta$ now assigns a distribution on a single state given the states at the child nodes. The behaviour of $A$ is now given by

$$\|A\|(t) = \sum_{\rho: \text{pos}(t) \to [0,1]} \prod_{\rho(x) \in F} \delta(\rho(x), t(x))(\rho(1), \ldots, \rho(n_x)).$$

Using this automata model, we can show the desired expressive equivalence:

**Theorem 2** Let $f : T_\Sigma \to [0,1]$. The following statements are equivalent:

1. $f = \|A\|$ for some probabilistic bottom-up tree automaton $A$

2. $f = \llbracket \varphi \rrbracket$ for some probabilistic MSO sentence $\varphi$

Moreover, there are effective translations between automata and logic sentences.

**Conclusion**

We have seen two new formalisms to describe probabilistic tree functions. Whereas probabilistic regular tree expressions capture the behaviours of the well-known probabilistic (top-down) tree automata, this automata model is not powerful enough to subsume the semantics of all probabilistic MSO sentences over trees. Therefore, we defined probabilistic bottom-up tree automata to obtain an automata model expressively equivalent to probabilistic MSO logic.

Current research focuses on a similar result for infinite trees and probabilistic Büchi tree automata [2]. Whereas generalising probabilistic regular tree expressions to infinite words seems to be promising, there already are fundamental problems defining probabilistic MSO for infinite trees.
References


Uniformization of Automatic Tree Relations by Top-down Tree Transducers

Sarah Winter (winter@automata.rwth-aachen.de)

Lehrstuhl für Informatik 7, RWTH Aachen, Germany

1 Introduction

The synthesis problem asks, given a specification that relates possible inputs to allowed outputs, whether there is a program realizing the specification, and if so, construct one. A related notion is the one of uniformization of a (binary) relation, which is a function that selects for each element of the domain of the relation an element in its image. The synthesis problem asks for effective uniformization by functions that can be implemented in a specific way.

Specifications are usually written in some logical formalism, while the uniformization, in particular in the synthesis setting, is required to be implemented by some kind of device. Since many logics can be translated into automata, which can also serve as implementations of a uniformization, it is natural to study uniformization problems in automata theory. Relations (or specifications) can be defined using automata with two input tapes, and uniformizations can be realized by transducers, that is, automata with output.

A first uniformization result in such a setting has been obtained by Büchi and Landweber in [1], who showed that for specifications over infinite words in monadic second-order logic, it is decidable whether they have a uniformization by a synchronous transducer (that outputs one symbol for each input letter). The specifications considered in [1] can be translated into finite automata that read the two input words synchronously. Such relations are referred to as automatic relations over finite words, and as $\omega$-automatic relations over infinite words.

The result of Büchi and Landweber has been extended to transducers with delay, that is, transducers that have the possibility to produce empty output in some transitions. For a bounded delay decidability was shown in [5], and for an unbounded delay in [4]. In the case of finite words, it was shown in [2] that it is decidable whether an automatic relation has a uniformization by a deterministic subsequential transducer, that is, a transducer that can output finite words on each transition.

Our aim is to study these uniformization questions for relations over trees. Tree automata are used in many fields, for example as tool for analyzing and manipulating rewrite systems or XML Schema languages (see [3]). Tree transformations that are realized by finite tree transducers thus become interesting in the setting of translations from one document scheme into another [7]. For a class $\mathcal{C}$ of tree relations and a class $\mathcal{F}$ of functions over trees, we are interested in a procedure that decides whether a given relation from $\mathcal{C}$ has a uniformization in $\mathcal{F}$.
2 Contribution

In [6], we focused on uniformization of automatic tree relations over finite trees by deterministic top-down tree transducers. In particular, we considered automatic tree specification that are deterministic top-down tree automaton-definable. We distinguish between two variants of uniformization. In the first setting, we do not require that a transducer validates whether an input tree is part of the domain of the given specification. We allow a transducer to behave arbitrarily on invalid input trees. In the second setting, the desired transducer has to reject invalid input trees. We speak of uniformization without respectively with input validation.

For uniformization without input validation, we consider the case that the transducer defining a uniformization has no restrictions. In particular the transducer is allowed to skip an unbounded number of output symbols thereby introducing delay. We showed that it is decidable whether a given relation has a uniformization by a top-down tree transducer, and if possible construct one. The question whether such a uniformization exists, is reduced to the existence of winning strategies in a safety game. The game is played between Player Input and Player Output, or short Player In and Player Out, where Player In can follow any path from the root to a leaf in an input tree such that Player In plays one input symbol at a time. Player Out can either react with an output symbol, or delay the output (a bounded number of times) and react with a direction in which Player In should continue with his input sequence. The winning condition expresses that Player Out loses the game if the input can be extended, but no valid output can be produced. However, in general it is necessary to skip the output an unbounded number of times. The key is that if the output delay exceeds a certain bound, then we can decide whether the uniformization is possible.

For uniformization with input validation, it turns out that this variant is more complex than uniformization without input validation. The reason for this is that in the employed transducer model it is not possible to verify the input without producing output. As a first result, we showed decidability in case that a transducer realizing a uniformization, synchronously produces one output symbol per read input symbol.

3 Outlook

For future research, we want to investigate whether a given tree automatic relation, i.e., a general non-deterministic tree-automaton-definable relation, has a uniformization by a deterministic top-down tree transducer. Furthermore, it is also of interest to study uniformization questions for different classes of relations and functions as presented here, as there are many models of tree transducers, see e.g. [3].

References


Uniformization of Automatic Tree Relations by Top-down Tree Transducers


Weight Monitoring with Linear Temporal Logic

Sascha Wunderlich (wunder@tcs.inf.tu-dresden.de)

Institute for Theoretical Computer Science
Technische Universität Dresden, Germany

Joint work with Christel Baier, Joachim Klein and Sascha Klüppelholz [2].

1 Introduction

In many application scenarios weight accumulation occurs rather naturally. One example is the total win or loss of a share at the stock market over one day. However, not only fixed periods of time are of interest. Formalizing more general time spans is often necessary, for example when considering the average CPU load within a specific computation phase. Many performance and reliability measures can be formalized using automata models with weight functions for the states or transitions. Resource requirements (e.g., bandwidth, energy consumption) and other quantitative system properties (e.g., the number of service-level violations) are then formally modeled as accumulated weights of path fragments.

We introduce a framework based on linear temporal logic (LTL) with the standard future and past temporal modalities and two new operators and . The latter impose constraints on the accumulated weight of path fragments τ satisfying a given regular condition formalized by a deterministic finite automaton (DFA) A. Side constraints formalized by nested formulae for the first and last position τ are possible. Formulae are interpreted over the paths of weighted structures. We consider finite-state weighted Markov decision processes (WMDPs). This includes weighted Markov chains (WMCs) and weighted transition systems (WTSs) as their degenerate forms without nondeterminism or probabilism, respectively.

Contribution

Besides the presentation of the syntax and semantics of linear temporal logic with weight assertions, we provide a reduction of its model-checking problem with bounded accumulation on weighted MDPs to the standard case of LTL model checking on MDPs. We also give decidability results for the unbounded case.

The main paper [2] contains details on the former approaches. It gives a more refined picture on the border of decidability and establishes sharp complexity bounds for different sublogics of our logic. Furthermore, it shows the relation of our logic to many recently proposed formalisms (e.g., [3], [4] and [8]) and finds some computationally easy patterns within the logic.

*The authors are supported by the DFG through the collaborative research centre HAEC (SFB 912), the cluster of excellence cAfAED (center for Advancing Electronics Dresden), the Graduiertenkolleg 1763 (QuantLA), and the DFG/NWO-project ROCKS, the ESF young researcher groups IMDa (100098198) and SREX (100111037), and the EU-FP-7 grant 295261 (MEALS).
\[(\zeta, k) \models a \quad \text{iff} \quad a \in L(\zeta[k])\]
\[(\zeta, k) \models \neg \varphi \quad \text{iff} \quad (\zeta, k) \not\models \varphi\]
\[(\zeta, k) \models \varphi_1 \land \varphi_2 \quad \text{iff} \quad (\zeta, k) \models \varphi_1 \text{ and } (\zeta, k) \models \varphi_2\]
\[(\zeta, k) \models \varphi_1 U \varphi_2 \quad \text{iff} \quad \text{there exists } h \geq k \text{ s.t. } (\zeta, h) \models \varphi_2 \text{ and } (\zeta, i) \models \varphi_1 \text{ for } k \leq i \leq h\]
\[(\zeta, k) \models \Phi^A(\varphi_1; \text{constr}; \varphi_2) \quad \text{iff} \quad \text{there exists } h \geq k \text{ s.t. } \text{trace}(\zeta[k...h]) \in L(A), \zeta[k...h] \models \text{constr} \quad \text{and} \quad (\zeta, k) \models \varphi_1 \text{ and } (\zeta, h) \models \varphi_2\]
\[(\zeta, k) \models \Theta^A(\varphi_1; \text{constr}; \varphi_2) \quad \text{iff} \quad \text{there exists } h \leq k \text{ s.t. } \text{trace}(\zeta[h...k]) \in L(A), \zeta[h...k] \models \text{constr} \quad \text{and} \quad (\zeta, h) \models \varphi_1 \text{ and } (\zeta, k) \models \varphi_2\]

Figure 1: Semantics of LTL[\(\Phi, \Theta : AUT\)] over infinite paths \(\zeta\) and position \(k \in \mathbb{N}\)

2 LTL with monitored weight assertions

For brevity we leave out some of the preliminary concepts. They can be found in the main paper [2], further details are available, e.g., in [1].

2.1 Syntax

To define a linear temporal logic with monitored weight constraints, we fix a signature consisting of finitely many weight symbols \(wgt_1, \ldots, wgt_d\), a finite set \(AP\) of atomic propositions and a class \(AUT\) consisting of deterministic finite automata (DFA) over the alphabet \(2^{AP}\). The logic LTL[\(\Phi, \Theta : AUT\)] extends LTL by two new modalities \(\Phi\) and \(\Theta\) to formalize constraints on the accumulated weight of path fragments. Formally, the syntax of LTL[\(\Phi, \Theta : AUT\)]-formulae \(\varphi\) is defined as

\[\varphi ::= \quad tt \quad | \quad a \quad | \quad \neg \varphi \quad | \quad \varphi_1 \lor \varphi_2 \quad | \quad \varphi_1 U \varphi_2 \quad | \quad \varphi_1 S \varphi_2 \quad | \quad \Phi^A(\varphi_1; \text{constr}; \varphi_2) \quad | \quad \Theta^A(\varphi_1; \text{constr}; \varphi_2)\]

where \(\text{constr}\) is a boolean combination of basic weight constraints \(\text{expr} \bowtie c\) for \(c \in \mathbb{Q}\) and \(\bowtie \in \{<, >, \leq, \geq\}\). A weight expression \(\text{expr}\) is a linear combination of weight symbols \(wgt_i\), i.e. for \(a_i \in \mathbb{Q}\):

\[\text{expr} = \sum_{i=1}^{d} a_i \cdot wgt_i\]  

(1)

We also introduce the notation PL[\(\Phi : AUT\)] for the propositional fragment of LTL[\(\Phi, \Theta : AUT\)]. The temporal operators \(U\) and \(S\) are disallowed here, the modality \(\Theta\) is unnecessary. Furthermore we denote the class of acyclic DFA by \(\text{Acyc}\) and the class of all DFA by \(\text{All}\).

2.2 Semantics

Formulae are interpreted over directed graphs equipped with a \(d\)-dimensional rational weight function \(wgt = (wgt_1, \ldots, wgt_d)\) and a labeling function \(L : S \rightarrow AP\) for a
set of nodes $S$ and atomic propositions $AP$. In particular, we deal with the MDP-semantics of $\text{LTL}[\Phi, \Theta : \text{AUT}]$ and interpret formulae over the infinite paths of a WMDP $\mathcal{M} = (S, \text{Act}, P, AP, L, \text{wgt})$, where $\text{Act}$ is a set of action names and $P : S \times \text{Act} \rightarrow \text{Dist}(S)$ assigns to each state-action pair a probability distribution for its successor states. Here, the weight functions are of the type $S \times \text{Act} \rightarrow \mathbb{Q}$.

The interpretation of $\text{LTL}[\Phi, \Theta : \text{AUT}]$-formulae is defined over pairs $(\zeta, k)$ of infinite paths $\zeta = s_0\text{act}_0s_1\text{act}_1 \cdots \in (S \times \text{Act})^\omega$ in $\mathcal{M}$ and positions $k \in \mathbb{N}$. As shown in Fig. 1, the semantics of the LTL-fragment are as usual. For $(\zeta, k)$ observing $\Phi^A(\varphi_1; \text{constr}; \varphi_2)$, there needs to be a position $h \geq k$ such that the trace of the path fragment $\zeta[k \ldots h]$ is accepted by the automaton $A$, the formulae $\varphi_1$ and $\varphi_2$ hold in its first position $k$ and last position $h$, and the weight constraint $\text{constr}$ is met. To determine this, each weight expression $\text{expr}$ is evaluated over the path fragment, i.e., for $\text{expr}$ as in Equation 1

$$\zeta[k \ldots h] \models \text{expr} \bowtie c \iff \sum_{i=1}^{d} a_i \cdot \sum_{j=k}^{h} \text{wgt}_i(s_j, \text{act}_j) \bowtie c$$

The satisfaction relation for finite paths and weight constraints is now defined in the obvious way, i.e., as a Boolean combination of basic weight constraints.

### 2.3 Examples

To illustrate the expressiveness and usefulness of our formalism we provide some examples. The used shorthand notations are defined in the obvious way, the operator $\Box$ is the usual LTL always operator derived by $\Box \varphi = \neg (\tt U \neg \varphi)$.

The following formula specifies that after each request a utility value of at least $c$ is guaranteed during some computation formalized by $A$:

$$\Box (\text{request} \rightarrow \Phi^A(\text{utility} \geq c))$$

To specify that the load during a some computation is between $c_-$ and $c_+$, we can use the formula

$$\Box \Phi^A(c_- \leq \text{load} \leq c_+)$$

More complex situations can be specified using nesting. The formula

$$\Phi^{A_{\text{init}}} (\tt; \text{energy} < c_e; \Phi^{A_{\text{work}}} (\text{utility} \geq c_u))$$

stands for the requirement that there is an initialization process which uses not more than $c_e$ energy and is followed by a working phase which in turn generates at least $c_u$ utility. The phases are formalized by $A_{\text{init}}$ and $A_{\text{work}}$, respectively.

### 2.4 Model-checking Problem

In the following we consider the complexity and decidability of the model-checking problem. For a $\text{LTL}[\Phi, \Theta : \text{AUT}]$-formula $\varphi$, the (qualitative) probabilistic model-checking (PMC) problem for weighted Markovian models $\mathcal{M}$ asks whether the (maximal) probability for $\varphi$ from an initial state $s_i$ is positive. Similarly, the (existential) model-checking problem for WTS asks whether there exists a path from a state $s_i$ fulfilling $\varphi$. 

271
3 Model Checking $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$

3.1 Reduction to the LTL-PMC problem

The goal is to provide a reduction of the $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$-PMC problem to the LTL-PMC problem. Given an $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$-formula $\varphi$ and a WMDP $M$ the idea is to replace all weight assertions $\Phi^A(\varphi_1; \text{constr}; \varphi_2)$ and $\bigtriangledown^A(\varphi_1; \text{constr}; \varphi_2)$ by an until or since formula, while adding information on the possible runs of $A$ for the path fragments in $M$. To that end, we enhance each state $s$ with a partial function $f$ for each occurring automaton $A$. The function tracks all the states $q$ the automaton $A$ can possibly be in after reading the trace of a path fragment ending in $s$, along with a vector $\vec{w}$ of the accumulated weights along this fragment.

This transformation generates an MDP which is polynomial in the size of the WMDP $M$, but (single) exponential in the size of the $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$-formula $\varphi$. The according rewriting of $\varphi$ into an LTL-formula is polynomial. The details of this construction as well as a proof of its soundness and its complexity can be found in the main paper [2].

3.2 Complexity

We now discuss the complexity of the model-checking problem for $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$ over WMDPs, WMCs and WTSs. The model-checking problem for standard LTL is known to be 2EXPTIME-complete for MDPs and PSPACE-complete for Markov chains and transition systems. These lower bounds obviously carry over to our logic, which is a superset of LTL. Using the reduction in the last section together with the automata-based approach of [9] for the LTL-PMC problem we obtain a 2EXPTIME algorithm for WMDPs. Furthermore, an adaptation of the approaches in [5] and [7] yields PSPACE bounds for WTSs and WMCs. Hence:

Theorem 1 The model-checking problem for $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$ is 2EXPTIME-complete for WMDPs and PSPACE-complete for WMCs and WTSs.

4 Unbounded Weight Assertions

So far, we studied the model-checking problem for $\text{LTL}[\bigcirc, \bigtriangledown : \text{Acyc}]$. The accepted languages of acyclic DFA are bounded in length, so we restricted the accumulation to path fragments of bounded length. Dropping this restriction leads to undecidability. This is well known and a direct consequence of undecidability results from the literature, e.g., in [3] for LTL with prefix-accumulation, which can be seen as a sublogic of $\text{LTL}[\bigcirc, \bigtriangledown : \text{All}]$. However, if we restrict ourselves to a single non-negative weight function, we get decidability even for the full logic:

Theorem 2 The $\text{LTL}[\bigcirc, \bigtriangledown : \text{All}]$-PMC problem is decidable for WMDPs with a single non-negative weight function.
Weight Monitoring with Linear Temporal Logic

<table>
<thead>
<tr>
<th>General</th>
<th>1-dim. non-negative wgt function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PL[ϕ : Acyc]</strong></td>
<td>NP-complete (see [2])</td>
</tr>
<tr>
<td><strong>LTL[ϕ, ⊸ : Acyc]</strong></td>
<td>WTS, WMC: \textbf{PSPACE-complete} (Thm. 1) \newline WMDP: \textbf{2EXPTIME-complete} (Thm. 1)</td>
</tr>
<tr>
<td><strong>PL[ϕ : All]</strong></td>
<td><strong>undecidable</strong> (see [2]) \newline decidable</td>
</tr>
<tr>
<td><strong>LTL[ϕ, ⊸ : All]</strong></td>
<td><strong>undecidable</strong> \newline decidable (Thm. 2)</td>
</tr>
</tbody>
</table>

Table 1: Decidability and complexity of the model-checking problem.

5 Conclusions

We established sharp complexity bounds and investigated the border of decidability for the model-checking problem of our new logics. Our main results are depicted in Table 1, where we distinguish the general case with arbitrary rational weight functions from the case of a single non-negative weight function. Both restrictions are necessary for the latter case. The NP-completeness for \(PL[ϕ : Acyc]\) and the undecidability for \(PL[ϕ : All]\) are taken from the main paper [2].

Even though the stated complexity bounds seem to make practical applications unfeasible, there are many techniques to make LTL model checking for MDPs applicable to real-world scenarios. Such methods are already in use by popular model-checking tools, e.g., an evaluation for PRISM can be found in [6].

References

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2012-01 Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04 Marcus Gelderie: Strategy Machines and their Complexity
2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

275
2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations


2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods


2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM


2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation
<table>
<thead>
<tr>
<th>Year</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-07</td>
<td>André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung</td>
</tr>
<tr>
<td>2013-08</td>
<td>Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika brahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers</td>
</tr>
<tr>
<td>2013-12</td>
<td>Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs</td>
</tr>
<tr>
<td>2013-14</td>
<td>Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures</td>
</tr>
<tr>
<td>2013-20</td>
<td>Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models</td>
</tr>
<tr>
<td>2014-02</td>
<td>Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software</td>
</tr>
<tr>
<td>2014-03</td>
<td>Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide</td>
</tr>
</tbody>
</table>
2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata

2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic

2014-06 Esther Horbert, Germán Martín García, Simone Frin­trop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video


2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs


2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
* These reports are only available as a printed version. Please contact biblio@informatik.rwth-aachen.de to obtain copies.